Ending Injection Vulnerabilities

CRAIG FRANCIS: [00:00:00] And today we're going to talk about ending injection vulnerabilities. We're going to briefly cover injection vulnerabilities, we'll talk about how taint checking can help or not, a special type of string, handling some oddities, examples you can use today and Go, Node, JavaScript, Java, and PHP, and also cover the future. And what we can hopefully be able to do in, let's say, 10 years time. The main thing we're going to be talking about is this quote from Mike Samuel, which is, "Distinguishing strings from a trusted developer, from strings that may be attacker controlled." By using this distinction, we can actually stop injection vulnerabilities from happening. This is based on the work from Christoph Kern, who did a talk called Preventing Security Bugs through Software Design. This was a USENIX Security 2015, and AppSec California 2016. The link is the link to the second talk, which contains a few extra details. It also involves information from the book created by some Google engineers called Building Secure and Reliable Systems. Just released in March 2020, and the section if you were going to read along is under the Common Security Vulnerabilities chapter, starting at page 266. Also like to give thanks to Toby Fox for our lead actors. Undyne, defender, and Spamton, our attacker.

CRAIG FRANCIS: [00:01:36] Ok, so injection vulnerabilities as we are a OS, we probably know these bit too well. But anyway, we're going to start with a very simple select statement for our database, and the developers incorrectly appended user data onto the end. Developer expects this to work and it kind of does work most of the time. The problem comes when the attacker comes in and starts appending their own SQL onto the end of it. So in this case, we're taking from the user table and we're saying where the ID is equal to minus one, which probably won't return anything. And then appending onto the end of that query, we're using Union to select additional records from the admin table, and we probably don't want Spamton, our attacker, to read that information. This is how are we supposed to do it today with parameterized queries where you would write the SQL as a developer defined string, you send that to the database. The database parses it and creates its query execution plan. As a separate step, the question marks in this case are replaced with the user data, which is the IDs they come in second, and that works. Great. Another approach that we can use is database abstractions. This is where developers don't really want to be writing their own SQL. Doesn't scale. It's a bit complicated, and the database abstractions can make

things easier. In this case, we're going to be looking at a very simple query, which was looking at the articles table, and we just see it simply saying we want to return the articles where the author is variable I.D.

CRAIG FRANCIS: [00:03:28] Another approach, you know, same API, is you would be saying, OK, where the author ID is null and you notice there's no user input on this one. You can also do things like functions, function calls. So this is not the most efficient way of doing it because it kind of breaks the index. But by using the date function, you can remove the time from the published field and then you'll be able to return articles that are published on a particular date. And that's all good. Let's take this a little bit further, and we're starting off with a query which simply returns articles with the word count of greater than a thousand. You could imagine this being here for listing the articles, which are long, you know, the long form articles, but what defines a long article might depend on the person. So there may be a request to come in to say, Well, I want to be able to customize this. The correct way of doing it, if you understood the database extraction correctly, you'd read all the documentation, you weren't making any mistakes, is you would provide them as two separate values again. First of all, you would say word count is greater than. That's the developer to find string. And then as a separate part, you provide the count. Again, nice and simple. But the sort of mistakes that can happen can be like this where they haven't provided-- the developer hasn't provided it as two separate arguments and instead used concatenation.

CRAIG FRANCIS: [00:05:09] Now, the database obstruction has no idea this happened. It's just going to accept, but because it came in as the first argument that it can be trusted and puts it straight into the SQL, and this is the main problem because the library code is usually well-tested, well-understood and works perfectly. But the vast majority of problems are actually due to the developer using the library. It's the input to the library and that's where we need to sort of focus. In this case, we can give you a quick example where Spamton has just simply not given a count, but has done, you know, is word count greater than word count, which is false. You can't have something that's greater than itself, and then it's doing the classic union and then selecting additional records from the admin table. Another example is a database abstraction that allows you to specify the order as just a single argument. And this does actually come up quite often because you often have tables of results which are sortable by the user, and it's so much easier just by just simply providing the value that is in, let's say, the

URL straight into the database abstraction. And because ordering doesn't necessarily refer to a single field as in in this case, the first name followed by the last name it gets, it's not really properly checked a lot of the time.

CRAIG FRANCIS: [00:06:42] So in this case, the developer is expecting the order by name first, followed by name last. But Spamton can do this. Now I'm going to give you this as a bit of fun, really, because order by is quite a difficult one to exploit because after you get to the order by part, you can't append additional records via union, but you can do this. You can alter the way the sorting happens. So in this case, we are trying to select the number one from the admin table when this condition passes it, simply looking at the admin table at record six and saying, Does the password begin with the letter A? If it retaining returned one, then the order by becomes ID equals one, and therefore the record which has an ID of one in this case, Amy moves all the way to the bottom of the table. If it doesn't match, Amy stays at the top of the table, so Spamton can issue lots and lots of queries. So this time now checking for password beginning with B. Imagine that Amy moves to the bottom and goes, Ah, bingo, I've got the first character of the password field and then move on to the second character. Obviously, we shouldn't be storing our passwords in plain text. And if you are interested in passwords, there's a brilliant conference run, typically every year on passwords. Password Com. That's two to three days worth of talks on this stuff.

CRAIG FRANCIS: [00:08:13] It's really fascinating, but I think we should move on. Ok. HTML. This is the classic example. You simply are introducing user data into the HTML. Spamton comes along and it simply adds a script hack. In this case, the evil alert. Ideally, we'd have additional mitigations in place, like a content security policy to block it. But there can be ways around that. So what we should be doing instead is using a library to build our HTML. The library has understanding of the HTML. It knows how to escape values and understands the different context those values can appear in. So the HTML will typically come from a static file or, in this case, a string. This is more common if you're using small snippets of HTML. You provide that template, that string, to the rendering, the template engine and then you provide the user details separately. It can then go away. And encode those values, according to the context. And that's great, and that works nice and simple. And you do have to make sure you templating engine is context aware. Otherwise, there are risks associated with that. Moving on from HTML, we also have command line injection. Again, another simple one user data is included

directly in the command. This fails if, because it hasn't been escaped, so Spamton can then write this where it's grep of a blank string and then providing a path to the secrets file and then the rest is commented out.

CRAIG FRANCIS: [00:10:05] So that doesn't actually get run. And it means that now Spamton is effectively returning all the lines from the secrets file. So how do we stop mistakes? So there's a thing called taint checking that could help here. This is where variables note if they are tainted or untainted. The most simplest version of this is you have a string which is seen as untainted, it gets assigned to a variable and that in this case, the HTML variable is now marked as untainted. If you have a slightly longer version, it's the different parts that make it up. So you're starting off with an untainted string. Your tainted variable, in this case name, and we're going to assume that came from a database or a query string post data or a cookie. And then you've got your untainted finishing closing P tag. Because this HTML variable now contains tainted data. It is also seen as tainted, and it means that our defending system, in this case, the HTML templating engine can now reject it. This is a similar example, but we're now going to use an escaping function. So you've got the tainted and untainted things. We're going to escape it to make it safe and therefore it kind of is safe. You know, it has been encoded correctly for that context. Brilliant. And therefore, HTML was seen as untainted.

CRAIG FRANCIS: [00:11:41] But unfortunately, taint checking incorrectly assumed escaping makes a value safe for any context. And that's a problem. It creates a false sense of security. So in this example, we have our same setup as before. We're going to be escaping, is this safe? Well, no. That didn't need to be any encoding, but because the a href now includes Spamton's evil alert javascript. It means that Spamton now has a cross-site scripting vulnerability. Ok. How about this one? It's an image tag. It's got a source. Sources can't include JavaScript, so kind of safe, so use your set up again. Is it safe? Well, we're missing some quote marks around the attribute and therefore it is not safe. There was nothing in that string that Spamton provided which needed to be altered to be encoded. So by providing a forward slash, the browser returns the home page for the image, which of course goes wrong. And then the on error attribute fires and then the alert is then executed. Ok, so this is a trick question. We are now going to put quotes around it. So same again, is it safe? No. Now, I say it's a trick question, because specifically in PHP before 8.1, which hasn't been released yet, single quotes

are not encoded by default. Now I fortunately got that changed beginning of this year. So if anything breaks, please let me know. But if anything breaks, you've probably got a problem.

CRAIG FRANCIS: [00:13:33] But anyway. Ok. And another quick example with the SQL, with my SQL in particular, with the real escape string, because obviously, you know, it's got to be real. There's the tainted, untainted. Now it's been escaped. Well, is that safe? No. Again, we're missing quote marks. The escape string is for escaping the string quoting character. Typically, the single quotes or the double quotes. Spamton hasn't used any of those characters, Spamton has just simply said minus one and then the union part is before. So this is also vulnerable to injection vulnerabilities. Taint checking is close, but escaping should be done by a third party library, and we can simplify it by looking for strings from a trusted developer. We can basically shorten it to safe versus unsafe. Safe, specifically, when talking about injection vulnerabilities, a string defined by the programmer in the source code and unsafe being everything else. So we go back to our examples from earlier in the prepared query. So let's start from users where ID equals question mark, is a developer defined string that is said it's safe. The ID is provided separately and we don't care about that. That's user data that's supposed to be kept separate. That's good. If we did append the user data to the SQL, that's safe, unsafe and therefore the SQL string itself is now considered unsafe. And then the database abstraction can reject this. You can say no, not having that same with HTML templating, safe, unsafe values being provided.

CRAIG FRANCIS: [00:15:24] That's good. As soon as you include user data into it, you know we are going to reject it. Same as before. And likewise, with the command line stuff. Here's just string from the developer, the values which come in from the user. That's good. If it's done incorrectly, then the command can't reject it. The same approach can be used from the database abstraction when the values are kept separate. This is great. You've got the published date is greater than the thing. These are kept separate. I'm just pulling this one out again because I do need to stress the definition of what safe is in this context. Remember, it's safe when talking about injection vulnerabilities. Spamton could give you a dodgy value or a value you weren't expecting. Now, in a select statement that's probably not necessarily the worst thing this could be a bit more important was a delete were to do everything based on this. So we're published is greater than year zero. Perhaps a bit more obvious when you're

talking about rm commands, because, you know, again, this is technically safe from an injection point of view. You know, we're not really, or we can't cover things which the developer is allowing to happen. So this is only safe again for injection vulnerabilities. Spamton could set the path to just simply saying forward slash and then you get your classic rm-rf joke.

CRAIG FRANCIS: [00:16:56] Obviously, a lot of our commands now prevent this, or at least ask you to provide an initial argument, but there we go. OK, so handling special cases. This is quite a common one, which is where you have the in clause and you will provide it with an array of numbers or integers, doing it this way is very convenient. It's quite simple. You notice I'm just using one function, the implode. But it's not ideal because did you remember to ensure that they are all integers? When you start talking about large projects mistakes are very easy to make, and if you look for example at the WordPress code base they use where in quite a bit and there are a few cases where they're not actually casting it to an integer. They have additional protections in place, so it's not actually a vulnerability, but we're getting into that territory of mistakes could be made. So, you know, in this case, we're not converting to an integer and then we have our where ID minus one so that we don't get anything from the first table and you've got your union from the admin table again at the end. How can we deal with this? Well, we use parameters again, as we should be. This time, we're just going to simply provide the right number of question mark parameters in there by just simply counting up the number of IDs that we want. That function, in parameters count, can just be as simple as building up and starting off with one question mark and then appending more as you need them.

CRAIG FRANCIS: [00:18:47] This is perhaps the easiest to read and understand in the context of a program or a defined string. We could get a little bit more fancy. And you can build up an array of question marks and then you can join or implode those values together and return that. Same as always, you've got to be careful with no IDs because databases don't really like it when you say where ID is in open bracket, closed bracket. But you shouldn't be running the query anyway, because it's not going to return anything. The second problem you might face is field names or table names. These can't be parameterized because they need to go into the original SQL because the database needs to create its query execution plan. You can't just simply append the value from the user, and it's dangerous, but you could escape the field or ensure it

matches a certain pattern. This is still risky, but because you shouldn't really allow them to just choose any field, you know, you could imagine a listing of users to a website where it's just simply showing their name and their profile. But if the attacker was able to search-- sort by their email address, the attacker can keep changing their email address and then sorting by the email address and seeing where they appear on the list. And from that, you can infer the values from other fields in the database, which they probably shouldn't be able to access.

CRAIG FRANCIS: [00:20:24] How do we deal with this? Well, we have to have an allow list, as in only these fields are allowed to be sorted by. This is good practice anyway, irrespective of this talk. So you have your allowed list. You use the array search function to get back the ID of that element. So what, you know, the order field is defined by the user and we use what the user said they want and we look through that array if it matches. We get back an ID. If it doesn't match, we get a false-ish thing, which is kind of zero and then we can use that array to then pull back the programmer defined string, so we are simply appending on the programmer defined string, and that means that we are still working with the same principle and we are also limiting to what fields we allowed to sort by. All good. What about config values? So things like tables, names which are set in INI, JSON, YAML files? Well, we'll have to cover that in the next section because it's going to be down to the library to deal with it. And the library is anyone that knows how to deal with these safely, as in they kind of need to be doing this anyway.

CRAIG FRANCIS: [00:21:41] Ok, ending injection vulnerabilities in Go. I'd like to thank Dima for checking over the code and Roberto for writing up the way in which Go HTML works. This is an example library that can be written in Go today. First of all, the library would create a type and we're just going to call it string constant. You notice it's got a lowercase S. So therefore it is not exported. It also then provides a method to call this library. And you'll notice that it has a capital O, which means it is exported. Its first argument has it actually uses its string constant and that is required for its input. So I put that to the side and then the actual code from the developer. This bit, we can kind of ignore it basically just returning the name from the user saying, you know, what's their name? That's our untrusted data. When you actually use our method from the library, the only accept string constant method, in this case, we are providing it with an untyped string. It is the developer defined constant. Go uses type conversion during compilation to turn that into the string constant. So even though it's not exported by the package, it

can actually still be used. Next, we have our example where the developers made a mistake. They are using the variable from the user. That cannot be converted to a string constant because it's a standard variable at compile time still unknown, and therefore it is rejected by the compiler.

CRAIG FRANCIS: [00:23:18] Brilliant. That's it. That's it sorted. So how did this actually get used by real package in this case? Go safe HTML or to give it the full URL. We're going to, first of all, look at the JavaScript part of this. It's the simplest one to use the script from constant method that it provides expects the JavaScript to be written by the developer. Now, I personally think you shouldn't be using in-line JavaScript, but it's still there and this is a safe way to doing it because the developer wrote it. It's-- they're the one who's trusted. If a unsafe value was provided, something from the user and the compiler would reject it in the same way as shown previously. When it comes to the HTML side of things, with the we are kind of having to use the query builder pattern, but we're taking the HTML first and it's going to be from the template package must pass an execute to HTML and that does the check. The safe HTML, HTML escaped method actually takes the value and encodes it and returns a HTML object. And then likewise, we see we're just doing another thing which returns HTML drop from developer to find string. And then the HTML concat function adds those three things together and gives you the output, which is all correctly encoded. This approach is not content sensitive.

CRAIG FRANCIS: [00:24:48] The templating system is, but this kind of works. If the developer was to make a mistake and to start using those methods they shouldn't do by passing in the user value like this, compiler will reject it. And likewise, during the concatenation bit, if that non HTML value was provided in, that would also be rejected. Ok, so how we do this in Node and one day JavaScript. JavaScript introduced a thing called a template literal. This did kind of cause a bit of a problem from the security point of view because it made it easier to do the wrong thing as in including a value not being escaped. For example, that gets included straight away. And if that puts onto the page, you have a problem. But there is a new version, I say new, still a few years old, called tagged templates. What this does is it takes a function and it calls it and provides-- the JavaScript engine provides the template in an array-like structure. Basically, it takes that template and splits it up into the different component parts. And then the values are provided separately as additional arguments. And here you see from the console log output how it has broken it up and basically what you end up having is the first part is

your developer defined strings. And the second part is the values. That's it being used correctly. Unfortunately, we've seen the case where a developer has gone, That's a function. They call functions with brackets. So they've put the brackets around and then it complains that the first argument should be an array, so they just put an array structure around it. And this kind of broke the entire thing. And what ends up happening is the template engine would just receive the first string and no additional arguments, so we just went, Yeah, that's it. Pass it straight through. All sorted a little bit later was also another example where that first thing was actually provided as an array. Likewise, but again, you'll notice that the user value is in that first array, not the second. So to the rescue is a new feature called is Template Object. Not currently available in JavaScript, but Node does have a poly fill. How does it work? Well, it's just a very simple function that you pass in the first argument and you're simply saying, Is this a template object, which is an array-like structure? Ideally, you'd also check to make sure is that sure it is an actual array as well. But the idea is, if this is not true, then we throw an exception. So when we use the tagged template correctly, that's all good and everything's fine. But if you don't use it correctly now, you can throw an exception. Nice and simple. How this works in Node, well, first of all, you have to install that poly fill.

CRAIG FRANCIS: [00:27:53] Npm install is template-- template object will just install it. To use it, well, you just simply require the poly first and then you just call the function. Now I'm going to use console dot log just to show us through like a debug mode and you'll notice on lines 12, 14 and 16, that's the same three examples that I showed you earlier. And when this actually runs, we have basically the first one is true because that was used correctly, and those two other calls which were done incorrectly get marked as false because they are no longer template objects. This function is hopefully going to come to JavaScript soon. This is the proposal that's going through the TK Thirty Nine process, and I'd like to thank Christoph for working on this at the moment. And Mike Samuel, who we saw earlier, who started this project off. There is a different approach in Node, which is the Google Closure Library. You just install it with npm and you require it. We're going to do our usual asking the person for a name just so we've got some untrustable data and then we're using this Google dot string dot const from method. The idea is it's supposed to be especially a developer defined constant. The second one, when you're using Node doesn't actually do anything, it just passes it through. It doesn't check it. But if you were to download the closure compiler and then use the closure compiler to make yourself a JavaScript bundle of all your JavaScript,

then it will actually complain. It will actually pick up on this and go, No, that's not good enough, and it will stop the code from executing.

CRAIG FRANCIS: [00:29:43] Now, going back, we're going to go off on a bit of a tangent now. I'm going to talk about Trusted Types. Trusted Types is available today in JavaScript, and it protects against DOM based cross-site scripting vulnerabilities. And we're going to have hopefully an extension to this soon. As described at this URL. So Trusted Types as it works today, you add it to your content security policy and you're simply telling the browser, We want to use Trusted Types and we're going to use them for script because, you know, in the future, we might have other things like styling. When you do your normal JavaScript, as in, you're getting a reference to the DOM node and using safe methods like text content. That's absolutely fine. That's allowed if you use inner HTML, that's disabled by default. There are loads of syncs in the the DOM API, which are less than safe. They're quite problematic. And it's not just in HTML, it's things like a val, timeout, even a HREF attribute, because obviously you can put JavaScript into that. There are loads of unsafe things, so the way Trusted Types works is it basically disables them, but it allows you a way to check your values before they are assigned.

CRAIG FRANCIS: [00:31:06] It's like a firewall for protecting values. In this case, we just create ourselves a nice, simple object, our Trusted Type object, and we have to provide a few different methods, or at least one which is named appropriately for what's going to be useful. In this case, I'm doing the dangerous thing of just simply passing the values through, but ideally you would use something like DOM Purify to make sure you know the value is safe. There's a new sanitizer API that's been developed at the moment, or you might use reg expressions or things just to make sure it's matching a pattern which you know to be safe. The advantage of this approach is it means that ninety nine percent of your code is going to be not using the unsafe APIs but the times you do use it, use this method and it means that the auditor who is looking over your code can check it and focus in on the problems. To make sure your Trusted Type object is actually understood by the browser to be trustworthy we call a method as in we're saying we're going to create this policy, and here it is. The name of it has to match a value that's in the header as well. So we're not making up anything that's not been allowed. And actually to use it is pretty much the same as before, but you now need to call the method to basically pass it through that firewall-like method that will check it and

clean it and make sure it's all OK before it goes into the HTML again, mostly for auditor purposes.

CRAIG FRANCIS: [00:32:36] And this is it working correctly. But bringing it back to the actual core idea of this talk. All of that code was actually quite verbose. When you know that the string has been written by the developer, it has no user input and therefore it's OK. So one of the things that's being developed is the idea of having a from literal method, which takes a template that can't have any variables in it, and it just simply passes it through. It's a way of simplifying it. And this is how JavaScript is likely to get the idea of what is a developer defined string added to it. And a nice, simple way in addition to the previous bit. So Java, there is a labor code error prone again created by Google, and it runs extra checks at compile time. Use your dependency management system, and you just append it. All the instructions on their website. It does have a few dependencies, but the actual code, the actual JavaScript or Java code, it's quite simple. First of all, you import the compile time constant annotation and then you just use it and you just simply say, but this argument should be a compile time constant. You get your name, the untrusted data and then any time that sensitive function is called, the compiler will check that it is a compile time constant. If it's not, then the compiler will reject it.

CRAIG FRANCIS: [00:34:15] There is also a Google GWT, but I think this might be a abandoned project because they haven't even updated it to use https yet. But it has a method in there called safe HTML from Safe Constant, which I believe works in the same way. There is also a possibility doing it in C++ but I never quite worked out how to do this. The Building Secure and Reliable Systems Book has a single line quote of using a template constructor that depends on each character value in the string, but no examples, and I'm not entirely sure what that is. And likewise, Google have written up a document called Safe HTML Types Overview, and they have a small quote there where they talk about using from constant. But again, there's no real examples of how that works, so I've not been able to get there.

CRAIG FRANCIS: [00:35:08] Ok, so with PHP, we can use Static Analysis. We're going to start with Psalm. This is the first one to implement this, and I'd like to thank Matthew Brown for doing this. Psalm is easily installed with composer, has a few dependencies. To start your new project, you just initialize it. And in that process, just make sure that

the error checking level is set to three or stricter. That's when it starts checking the types for this particular time. So add just simply use, you specify that you will be using a literal string type and for the arguments as to how it actually works in code, basically, the Static Analysis tool will just pick up on the fact that when you've actually added in a user value into the literal string, it will reject it. PHP Stan is very similar, and I'd like to thank Ondrej for this. Composer require to install-- it just downloads. PHP sounds slightly different in that you have to be level five or stricter when there's only a single type as in your're only accepting literal strings. But it has to be seven or stricter when there are multiple types. For example, if you're accepting a literal string or an array and in this case, level nine is the most strict, but it works in exactly the same way. You just say the parameter type is literal string. And then when it actually picks up on the issue, you know it rejects it from that.

CRAIG FRANCIS: [00:36:44] Ok, so the future. Static analysis is used by about a third of developers, and that's good. You know, it's good that it's happening, but it's not really targeting the right people. Most developers who are probably creating the most injection vulnerabilities are probably not using Static Analysis. I'd like to thank Jet Brains for creating this survey, which was released earlier this year, but there is also a slight bias in this results.

CRAIG FRANCIS: [00:37:20] It's certain developers have filled out this form. Laravel is a popular framework, but on the wider system it's unlikely to be very representative because I'm fairly sure that WordPress does kind of take the lead on the number of developers in the wider world, and especially as WordPress is not easily managed with Static Analysis. So this is why we can't with the PHP and the is literal RFC. I'd like to thank Joe and Mate on this. Joe created the implementation. It is available, you can add it as a patch PHP 8.1. And Mate did performance testing on it to make sure that we weren't making the process slower. This is the website where we put the RFC. Unfortunately, it did fail as a through the vote, mostly because there was some communication problems. I wasn't very good at explaining myself and also we probably left it a little bit too late to actually do the vote, as in the vote was finishing on the last day before feature phrase. So perhaps quite understandably, people getting a little bit unsure about it. Reason I want to go about it in this particular way, it has no dependencies, so everyone will get it and libraries can start using it straight away. It is easy to use. It's a very simple function that allows you to just do this test.

CRAIG FRANCIS: [00:38:53] There's no need to use Static Analysis, but it works very well with it. You know, so the Static Analysis tool can actually use that function and then sort of infer a particular type and then work back from it. And that would be very useful. And this is probably the most important is it works with existing code and libraries. So a lot of libraries and a lot of code already exists, and we don't want to have to have everyone rebuild everything with query builders and things. You know, this is the the argument about, you know, do you support concatenation or not? If you don't support concatenation, it can help with debugging. But if it requires you to rewrite huge sections of code because, you know, and it's not going to actually improve your security by doing so, that's a lot of work, and therefore it probably won't help with adoption. So we've gone with this approach, which does support concatenation of literal values to aid adoption, to make it easier. And by using a function, it allows you to choose how to handle mistakes. So you can write to a file, you can write to a database, call an API, throw an exception or just simply do nothing. So libraries don't need everyone or-- the advantage of this approach is it means that libraries don't need everyone to understand and read all of their documentation, which is kind of the big problem because libraries get a bit complicated.

CRAIG FRANCIS: [00:40:23] There is a lot of methods they provide, and it's unlikely that everyone knows exactly what is, you know, which arguments are safe or not. And it also doesn't mean that we don't rely on developers never making mistake because even the most experienced of developers might make a mistake and not be able to notice it. And another thing is that developers will be warned as soon as they write their code. So when people talk about security being pushed left as in the closest to writing the code as possible, it helps the process. So, for example, an ID could highlight problems as the developer types or if they're not using an ID, you know, this happens as soon as the developer runs their code. So as they are writing the code, they can find out about the problems. Generally speaking, we found a 0.7 percent performance impact at a time when PHP was getting like a 30 percent performance improvement. We did test this on Symfony demo without connecting to the database. This meant that, you know, because if we were connected to the database, there's too much variability. So we were trying to find like the most pessimistic way of measuring performance. How it works? Well, first of all, we just checked to see if that function exists with backwards compatibility reasons. And then if it is a if it's not a literal, then we throw an exception.

CRAIG FRANCIS: [00:41:52] And as to how it actually is used, we simply call the method. And that means that every method that the library uses, it can just call the is literal check, which the private method. Is this too strict? Probably. So how about this? Have a protection level. Zero for no checks, one for just warnings and make that the default and then two for exceptions. So for those who really want to make sure. So if you're dealing with medical records and banking information, you probably want to go with exceptions, but most people will probably be fine on warnings. You may even have it so the exceptions are in development mode, but there are no checks on the production server. And the actual literal check that we showed earlier, expand it a little bit so that you do your does the function exist or is it a literal? That's good. Fantastic. We have a way of sort of saying, is this an instance of an unsafe value, which we'll talk about in a minute. And then you go through the different production levels, which would either trigger a warning or an exception. So that special case of an unsafe value. We have a simple stringable value object. It shouldn't be needed. This is how you probably use it if you did, where you just simply say new value-- new unsafe value and you put your string in there, which is unsafe, and then you can then just call in the DB query.

CRAIG FRANCIS: [00:43:21] This is the best example I can think of, but to be honest, you shouldn't be doing this. Think back to that order by example, where we had an array of allowed values and pulling from there. That's probably should have been a better approach. The other advantage of this one is it makes it easier for the auditors to find. So when I'm looking through code, it'll be really good to see, Oh yes, why are you using that?

CRAIG FRANCIS: [00:43:44] Ok, so how about identifiers in SQL. It's basically the same method where you have your query SQL and parameters, but identifiers which is your table names and your field names, the things which might be coming from the INI or the JSON or the YAMAL config, they get provided separately. So you do your first check on your SQL and then you go through the identifiers and you just very simply check, you know, is the identifier acceptable? And this is very strict, but it knows what the restrictions are, and therefore the library is able to safely include those identifiers, which are no longer literals. But they're applied after the check, and therefore it's perfectly safe. So then that gets into the database. How this looks and being used? Well, you do DB query as the method, if it's used correctly, where the first argument is a

developer defined string, that's good. As soon as you concatenate in some user data, then it gets rejected. And this is the example of using identifiers. Unusual, but it's possible. And therefore we are covering the idea of having variable identifiers. If you think back to that database abstraction where we had the where method where you could say field or value, it meant that you can sort of run the check on the literal check. And likewise, with the order by. It might have a different approach with order by with the list of allowed values.

CRAIG FRANCIS: [00:45:28] Command line interface version, you can have a parameterized exec kind of function where we would just simply check the base command is a literal and we're going to throw an exception on this one. We're going to be very strict. This is how you'd apply the parameters to that command. It's a very simple loop where we just escape shell arg, you know, the arguments get escaped and added to the command and then just simply run. This is how it would work correctly, as in the first bit, is that developer defined string. That's checked. That's all good. If it's included user data that is also checked but rejected. Because, yeah, same with HTML. You can render the HTML where you have the arguments provided that's checked and obviously encoded correctly. Good. If the user data included that would be checked and rejected. For a HTML templating engine, I was able to create one in about three hundred lines.

CRAIG FRANCIS: [00:46:34] You know, this is a context aware one. We start off with the protection level, as we discussed earlier. I have an array-like structure which defines which tags are allowed, the attributes and the the values for those attributes. I'm using HTML parsing with an XML mode because I don't know, XML creates a certain quality maybe. More importantly, though, it ensures that all the tags are nested correctly and all the attributes are quoted because you have to make sure you're actually being quoted. There's this little section here for walking the nodes to find the question marks for parameters and then the bit, which actually does the returning of HTML with the parameters that are now checked. And then at the end, I've included an unsafe value object. Shouldn't be needed, and so far I have not needed to use it. How this looks. Well, a bit like the templating engine before. I'm using question marks for the parameters of all the names, because in this case we were just using short snippets and therefore it's nice and simple. You can also do a template and then use it multiple times. That's good. And if the developer makes a mistake and includes the user data in that

first argument, it gets rejected. Exceptions fine. And also because it context checking when a normal URL is included, that is all good.

CRAIG FRANCIS: [00:48:00] If it's a JavaScript URL, it's rejected. And you know you can actually go a little bit further by having a href image source. They ensure they are URLs. Width and height are integers as according to the HTML spec, they are supposed to be. Alt and figcaption are text.

[00:48:21] So how would this work in, let's say, 10 years time? Once all of the libraries have started using this? Because you notice the libraries here are just checking their input, the output and specifically how the native functions like my SQL query and my SQL prepare work. Those are not being checked yet. Now how we go about this is very much up to debate. But theoretically, how about they accept everything, but they can warn, if not given a string from a trusted developer? However, there also needs to be a way for special cases to be trusted, e.g. strings created by a library. You'll notice this is very similar to the Trusted Types example we talked about earlier in JavaScript. That firewall-like structure. Maybe this can be done with a stringable value object. So that value objects we created earlier for the unsafe value, just rename a few things, and we'll instead have it a way of marking this SQL string as safe, and it still acts like a string. But now it's sort of wrapped in an object. How this could be used? Well, you'd have some way to say that this particular stringable object is trusted. For shortness I've just said it can be trusted for SQL. In reality, it's probably going to be more of a set way of saying, Oh, it's this argument for this method or this function, because what is safe in one system, they're not necessarily safe in another. So this is probably fine for my SQL, but might not be fine for PostgreSQL. When it actually comes to actually using it, the query method in this case will build up its SQL, which is no longer a literal, but it will wrap it in this DB trusted SQL value object. But because that value object has now been trusted, the my SQL I query function says, Yep, that's fine. I'm accepting that no warnings, no errors or anything because you've gone through that process. There would need to be a way to disable the check. But the beauty of this, it means that the developers are acknowledging the fact that they have a problem and sort of saying, Yeah, I know I'm doing an unsafe thing and that's up to them. Know that's their choice. There would need to be a way to enforce the check because some programs need that extra level of protection, and it would be good for developers who are confident their system to be

able to enforce it. Again, talking about medical records and banking data, anything like that. You might want extra level.

CRAIG FRANCIS: [00:51:07] But yeah, in short, distinguishing strings from a trusted developer from strings that may be attacker controlled. With that distinction, we can stop injection vulnerabilities. Thank you.