PROFESSOR: All right. So let's get started with the second lecture in our stunning series on web security. So to start off the class today, I actually want to go over some quick demos. So as you know, demos almost never work. So hopefully, you won't just be seeing my empty terminal up here. But the basic idea is that I first wanted to show you an example of the Shellshock bug that you may have heard of. This has been a pretty sort of popular topic in the security literature.

And people were saying that Heartbleed was like a 10 out of 10 security [? bug. ?] But people were saying, like, we should not have reserved 10 out of 10 for Heartbleed. This is potentially worse. All right? And so I thought that this would be a great idea for you guys to see some living history, and for you to tell your parents that, you know, they're getting their tuition's worth out of MIT.

So what is the basic idea behind the Shellshock bug? Well, it's a really great example of why it's so difficult to build secure web applications that span multiple technology stacks, multiple languages, multiple OS's, so on and so forth. So the basic idea is that Shellshock is going to take advantage of the fact that the attacker can craft a special HTTP request to a server and control the headers that are in that request. And so I've written an example up here. It's very simple.

So let's say that the attacker wants to send some GET query. They're going to send that query to some CGI interface. And then there's going to be some question mark. The person wants to search for cats, because that's all that people search for. And then there's some standard headers here, like host, for example. So this is saying that this URL here is hanging off of example.com.

Now, note that the attacker can also specify custom headers. Right? So the attacker can just say, I want it to find some application-specific header called Custom-header, and I want to specify some value there, because you can imagine that a web application might define certain functionalities that can't be expressed using the simple, pre-defined HTTP headers. OK. So that all seems fairly innocuous.

But what ends up happening is that in a lot of these CGI web servers, they will actually take these custom header values and use them to set environment variables for Bash. OK? So they will use this header to create a Bash variable name custom header. Then they will take this value here that the attacker has supplied, and use that to be the value of that Bash variable, right? And then once that variable is set up, then the CGI server will do some processing in the context of that environment. Right? So this is clearly bad. You can probably see where this is going. Web servers should not be taking these arbitrary values from arbitrary unwashed masses.

So in the particular example of the Shellshock bug, what ended up happening is that if you set your Bash variable to this, this kind of malformed, evil-looking thing, then there's going to be insanity that happens. Basically, this is a malformed [? Select ?] function definition in the Bash scripting language. You don't have to worry about the specifics of it. But what was intended to happen, if Bash were correct, is that this part over here wouldn't be executed. So basically, you just defined some stupid function here that doesn't do anything. And in the [INAUDIBLE] terminate here.

But this sequence of characters actually confuses the Bash parser. And so what ends up happening is that it sort of stumbles through this nonsense here. And then it says, oh, I might as well keep on parsing and execute some commands here, right? And so in this case, this just does the bin/id command, which displays some information about the user. But this could be any code right here. So that's the heart of the vulnerability.

So I'll give you a very simple example here, so you see up on the screen. So basically, we've got a very simple Python server here, just the dumbest one you could possibly imagine. It's got this do GET method. And so with the do GET method, it is going to basically iterate through all of the HTTP headers in the request. OK? So that's what this four key value for the header and the value in this request. And then it'll just print out the headers that it finds. And then in this dirt-simple example, it's going to do something very dumb, which is execute the system call and just directly set the environment value to the value specified in the header. So that's the whole root of vulnerability.

So if I come over here and I start my victim web server-- OK, so it's now ready to accept requests. And then I can write my special Shellshock client like so. And this is actually pretty dirt-simple. So here, I just define one of these malformed strings. So I have these kind of janky characters at the beginning. And then I know that everything after this is essentially going to be executed on my behalf on the server side. So in this case, I pick something that was actually pretty innocuous. It just says, echo, I own your machine. But this could be anything here. You could start another Bash shell kind of like I do here. And then, echo attacker command, where in the real world, that could actually be something very dangerous.

So then I set the headers and my custom request. And then I just use Python to create an HTTP connection and just send it to server. So what ends up happening? So I execute my Shellshock client here. So it's saying that I had a 404 here, because it doesn't matter what file I requested. So I just put in some index, an HTML that doesn't exist. But if we look over here, this is the output for the server. And so what you see is that you have this output, I OWN UR MACHINE, and ATTACKER CMD. And that's because as the server got that header, it set the Bash variable. It set it with this weird thing here. And as a result, an ATTACKER-controlled command got the run. So does that all make sense?

AUDIENCE: So does this happen if the program is run under that? I'm still unclear on, like--

PROFESSOR: Yeah. So the specifics of how the attack works actually depends on are you running Apache, like what exactly your web server looks like. So in this example, it's a little bit contrived, because I actually called [INAUDIBLE] explicitly spawned off another Bash shell, set the environment variable in there, and then we were ready to go.

But you could imagine that if you were spawning off a different process for each incoming connection, you could set the environment variable for that directly if that guy was using-- was living inside of a Bash environment.

AUDIENCE: So if you go back to your web server code, it seems that you have a much worse vulnerability than the Shellshock, because you're calling [? though ?] a system. And I can execute a command just by setting the custom header to something [? that I ?]. I wouldn't have to use the Shellshock bug in this example.

PROFESSOR: That's correct. Yeah. So in this particular web server, which is something I wrote just for sort of teaching value, yeah, this thing you shouldn't trust for anything.

AUDIENCE: But the Shellshock exploit was on assigning something malicious to an environment variable using [? set N ?] or something like that, which is something [INAUDIBLE].

PROFESSOR: Oh, yeah, yeah. So that gets back to his question. That's right. So if you had, like, let's say, Apache up here, Apache's a little bit tricky to sort of configure in a way that's obviously what's going on. But you're exactly right. So Apache would call Set nth, which is another way that you can directly set the environment value for whatever particular service [? I ?] process you have.

But you also actually have some servers like this that you can imagine that they actually do a spawn of a separate process and do something very morally equivalent to this. But you're exactly right, that the way that a patch in particular was violated was the way that you described. So does it all make sense? OK. So that's sort of a quick and dirty example of Shellshock stuff.

And so another example I wanted to give you was an example of a cross-site scripting. And so the Shellshock bug was sort of an example of how content sanitization is very important. So as we'd just been discussing, you shouldn't just take inputs from an arbitrary person and them use them directly in commands of any type. So cross-site scripting attacks are another example of how something can go wrong.

So in this example, I have another sort of dumb CGI server here. And if we look at this CGI server, so what is it going to do? So once again, I've written something very simple in Python. This is going to be the handle that executes when a request comes in from the client. And so essentially, what happens is that up here, I'm going to print some headers for the response. So I'm going to say, my response is going to be of type text HTML. This line here we'll actually explain in a second. So as it turns out, browsers have some security mechanisms to try to prevent the attack that I'm about to show you. So I put this example-- I put that header line in there to turn some of the protections off.

And then what the CGI script does is it gets access to all of the fields and the CGI requests. So imagine that everything in a query string after this question mark-- like these header and value things, that's what goes into that form example there. And so what the CGI script does is something very simple. It just directly prints the value of something that was passed from the attacker. So same basic idea. This is a bad idea, because this Print statement, it's printing directly into the HTML itself.

So what can happen is as follows. So let's say that I have a bunch of queries I want to run. So in this first query here, I'm just setting the message value to Hello. So if I go over here and I run that page, well, then you're going to see that this Hello shows up, because once again, the server was taking directly what I pass to it. And it prints Hello. So no big surprises there.

Now let's say I realize that I can actually pass arbitrary HTML in there. So now I actually try to embed some styling in there. So I say, h1 and then Hello again /h1. So that worked, right? So once again, we're printing directly into the [? pake. ?] So now you might think, OK, we're in business now. This is cool. So let's just directly embed some JavaScript code in there. All right. And so I do this. And here, I've actually just put in-- for the message, I put script. And then I want it to just alert XSS and then script. So now that's interesting.

So it seems like something didn't quite work. So I don't see any output. I didn't see the alert either. And if I actually look at the output for the web server-- and what I see is that here, the web server itself didn't actually get that trailing script tag. So it seems like the browser itself has somehow detected something evil even though I tried to disable the XSS filter. So that's interesting.

We're going to come to this defense mechanism a bit in the lecture. But suffice it to say, it seems like the browser is trying to resist this cross-site scripting attack. But of course, what we can take advantage of is the fact that HTML, and CSS, and JavaScript, they're extremely complex languages. And they compose in these very difficult to understand ways.

So here, this is what I've been setting my attack string here. This is malform URL. I'm saying, image, and then three quotation marks in a row, and then a script tag. Like, this shouldn't actually parse. But what's going to end up happening is that the browser's going to get confused here. So it's built-in cross-site scripting detection actually fails here.

And so what ends up happening is that now you see the alert. OK? And what's interesting is that if you actually look at the contents of the page now, it's kind of messed up. Like, where did this quotation mark and brace come in? If we do a Control, U, we can see that this does not make the browser happy in some way. That's a little bit unclear. But it doesn't matter if we're their attacker. We saw that alert. That means that [? our code ?] got the run. And from the perspective of the attacker, who cares that the page is messed up now? Because I could have used that code to steal the cookie or things like that. So does that all make-- yeah?

AUDIENCE: What's the cross-site aspect?

PROFESSOR: Ah. So the cross-site aspect is that if the attacker can convince the user to go to a URL like this, then the attacker's the one who's specifying that stuff in the message. It's the attacker who's specifying the alert XSS or something like that. And so essentially, what's happening is that the victim page is executing code on behalf of someone that is not that page.

AUDIENCE: Can you explain exactly what the browser roles are for sanitizing [? games ?] for [? play? ?]

PROFESSOR: Yeah, yeah. So we'll get to that in a second. So we'll get to that in a second. OK. So that is all for story time. And let's see here. So I guess I can turn this guy on. And maybe he will [INAUDIBLE] guy here. This guy here.

AUDIENCE: Front wall.

PROFESSOR: Ah, OK. There you go. All right. Eighth time's the charm. OK, thanks. OK. So yeah, so those are just two quick demos to sort of show you the filthy and dirty world that we live in right now. So why is cross-site scripting so prevalent? Why are these problems such a big deal? Well, the reason is that websites are increasingly more and more dynamic, and they want to incorporate a user content a lot of times, or they want to incorporate content from other domains. So think about the Comment section on a news article.

Those comments come from untrusted folks, from the users. So somehow, these sites have to figure out, what are the rules for composing those types of things? And also, the websites might host user-submitted documents, a thing like Google Docs or Office 365, for example. Those documents all come from untrusted folks, but somehow, they have to live with each other and with the large infrastructure from Google or from Microsoft or whatnot.

So what are some of the cross-site scripting defenses we can use? This kind of gets to your question. So we'll actually look at some of those now. So one type of defense is to basically have cross-site scripting filters in the browser itself. And so these filters will essentially try to detect when there's a potential cross-site scripting attack. And so we actually saw one of those filters in action. And I think that was the third example that we looked at.

If you have some website-- or some URL-- excuse me-- that looks like this-- so foo.com. And then you have some question mark and then some query string you're going to submit. This is very similar to the example that I tried third. So I just set this source to something like evil.com/cookiestealer.js. And so what ended up happening was that when I tried an example similar to this, the browser actually rejected it out of hand. So we saw that it didn't even work.

And the reason why it didn't work is because the browser looked and said, is there an embedded script tag in a URL? So basically, it's a very simple heuristic for figuring out if something evil's probably going on, because no legitimate developer-- or no developer that's sane-- should be doing stuff like this. So there's actually these configuration options in your browser you can use to turn these things on and off. Occasionally, this is useful for testing if you just want to inject some JavaScript really quick and dirty. But this is almost always assigned [INAUDIBLE].

So for example, Chrome and IE have a built-in filter that will look at your URL value in the address bar, look for things like this. And if it's there, they will do things like maybe delete this whole thing completely. They will maybe change the source to be empty, stuff like that. And so essentially, to get to your question, there's a bunch of heuristics that the browsers have to identify things like this. And if you look at the OWASP site, they actually collect examples of heuristics you can use to detect cross-site scripting, as well as tricks you can use to bypass those filters.

So it was very funny. So the first thing I wanted to do for the demo is do something like this, and it didn't work. So then I went to the OWASP cheat sheet. I looked at, like, the third thing they suggested, and the third thing they suggested worked, which was that sort of broken image syntax type stuff.

So the basic problem with just relying on this is that, like I said, there's a lot of different ways to force the CSS and HTML parsers to mal-parse something. So these things are not complete solutions. They don't have the perfect coverage.

AUDIENCE: Shouldn't this just be like the lead-in from the browsers? Because it seems like not the browser's job to do this.

PROFESSOR: You mean it's not the browser's job to sanitize this kind of stuff?

AUDIENCE: Yeah.

PROFESSOR: I mean, you could imagine sort of having a browser sit atop a proxy, for example. And maybe the proxy did sort of cleaning like this. I mean, intuitive reason why it might make sense to do it inside the browser is because so many of the legitimate parsing engines are inside the browser. So presumably, if you're closer to where the actual parsing's being done, it's better. But you're right. In practice, you can imagine there being sort of defense in layers, basically.

AUDIENCE: I think what he might be saying is that it's the web developer's job, not the client's job to sanitize this.

PROFESSOR: But, I mean, that's kind of like saying-- so in a certain sense, we could say that about processes, too, in Unix or Windows. So we could say it's sort of developers' jobs to make sure those things stay isolated. But in fact, the OS and the hardware as well has an important role to play, because [INAUDIBLE] trusted whereas any two arbitrary programs developed by any two arbitrary developers may or may not be trusted to sort of implement security correctly. But you're correct. And in fact, frameworks like Django or whatnot, they actually try to help you to get around some of these problems.

So anyways, so yeah, so filters are not a perfect solution. And also, filters can't prevent what's known as a persistent-- persistent-- cross-site scripting attacks. This is known as sort of a reflected or transient one, because this script code just sort of lives in the URL. Then once the user's closed that URL, basically, the attack's gone.

But you could imagine that you could have someone who-- user puts malicious HTML in the Comment section for a website. And if the web server actually accepts that comment is valid, then that comment, with this malicious payload, can essentially live there forever. So whenever any user goes there, they would be exposed to that malicious content.

Another example, which is sort of funny and sad, as all these things, is if you look at dating websites. So some dating websites actually allow users to put full-blown HTML in their profile. So what does that mean? So when someone else is lonely, presumably, or looking to find their one true soul match, they go to your website. They're going to run HTML that you've crafted in the context of their session. And so that can also be a very damaging attack as well. So just doing these kinds of filters don't protect against things like that.

AUDIENCE: So [INAUDIBLE] in the Comments section presumably does that by setting a post-- the information goes to the server in a post variable or something like that?

PROFESSOR: So there's a bunch of different ways you could imagine doing it. Yeah. So one way you could imagine doing it is a post. Another way you could imagine doing it is a dynamic XML HTTP request.

AUDIENCE: OK. But if it's like a post, why can't you just scan through it and do the same thing that you have in the--

PROFESSOR: Yes. So you're exactly correct about that, and we'll discuss some of that in a second. But you're exactly correct that the server side of the application should be very defensive and mistrustful of this stuff. So you're exactly right. So you could imagine that when the server maybe saw something like this, [INAUDIBLE] even if the browser did not. You're correct about that. All right. So that's basically a survey of these cross-site filters in the browser.

So another defense against cross-site scripting is something known as HTTP-only cookies. And so the basic idea behind this is that a server can actually tell the browser that client-side JavaScript should not be able to access a particular cookie. And so basically, the server can just send a header value in response in the set Cookie field. It can say, hey, don't let clients like JavaScript manipulate this cookie. So only the server could do this.

And so this is only a partial defense, though, because the attacker can still issue requests that contain the user's cookies. So this was the cross-site request forgery that we looked at in last lecture. So even if JavaScript code can't manipulate cookies, the attacker can still do things like conjure up a URL to some e-commerce site, let's say buy.com. The attacker can put whatever item the attacker wants to buy. So puts a Ferrari, for example. And then the attacker can then say, who should this go to? This should go to the attacker.

And so even though clients like JavaScript can't access the cookie, there's nothing that prevents the attacker from just conjuring up a URL like this. This is what some of the CSRF tokens help to prevent against, which we'll talk about a little bit later.

So another thing that you can try to do to prevent these cross-site scripting attacks is privilege separation. And so the idea here is basically that you want to use a separate domain for all the content that is untrusted. And so for example, a lot of the online server providers were things like email or online productivity suites. So think Google Docs, Office 365, so on and so forth. They actually use a separate domain to host user-submitted content.

So Google, I think they still use this. They used to put all the stuff that users submitted into some special domain called googleusercontent.com. And so here, they would put things like cached copies of pages, your Gmail [INAUDIBLE], and things like this. And at least as of a year or two ago, this is like one of the top 25 [? Alexa-visited ?] domains, because Google services were so popular.

And so what's the advantage of putting stuff in here? Well, the hope, at least, is that if there is some type of cross-site scripting vulnerability or something like this in a user-submitted content, then hopefully, the daemons would just be limited to that domain. It wouldn't actually affect the full-blown google.com. This isn't a perfect defense, though, because user-submitted content may have references to things from google.com. And so once again, this is only sort of a partial fix for a much more pervasive problem.

Now, another thing you could do-- and this gets back to the gentleman's suggestion over here is that we can actually do content sanitization. And so the idea here is that, essentially, whenever you-- where you can be the browser, where you can be the web server, or whatever-- whenever you receive untrusted content, you don't trust it at all. And so you go through it, and you do things to sort of render it sort of neutral such that it can't actually execute code or subvert your system in any way.

And so an example of this is the Django template system. And so Django is an example of a web framework. So basically, the high-level web framework is something that helps to automate and secure some of the sort of tedious tasks of developing a website. So it will help you with making database access easier. It'll help you with doing things like session management. And it will also help you with maintaining a consistent look and feel across your website.

And so one way to maintain that consistent look and feel is to use this notion of templates. So all of your pages automatically start out with the same CSS and things like that, the same styles. But then there's these portions in the web page where you can specialize it with the particular news article that's at the top of everybody's mind that day, or something like that, or user-specific content.

So for example, in Django, you can look at a template, and it might look like something like this. So you have a bold tag. It says, Hello. And then you have these braces here, these double braces. And it says, name. And so essentially, what this means is that this is like a placeholder variable. So essentially, these pages get dynamically generated. So when the user goes to a Django site, the Django server says, OK, well, this name is going to be somewhere, who knows, in the cookie.

Maybe it's going to be in a CGI string, whatever. And so as the Django server dynamically generates the page [? to return ?] to user, it replaces this special reference here with whatever the value of this variable is. So it's pretty straightforward. This is kind of like that dinky CGI server I showed you. So just reflecting user-submitted content right here.

But Django actually does it better than the silly CGI server that I showed you, because it uses this notion of content sanitization. So Django expects that users may be adversarial. So it's not just going to directly put the value of the name variable here. Instead, it is going to encode it in such a way that this content will never be able to escape out of the HTML context and execute JavaScript or something like this.

So for example, one thing it'll do is it'll take the angle brackets, and it will translate them into these HTML entities. So the less than character gets transformed into this. The greater than character gets translated into this. Double quotations get translated into ampersand quote, and so on and so forth. And so what this ensures is that if the content the user put in name actually tries to contain angle brackets or things like this, then it'll basically be neutered. And it'll be translated into something that would not be interpreted as HTML on the client-side browser. So does that make sense?

So now I know that this is not a completely foolproof defense against some of this cross-site scripting stuff. And the reason, as we showed in the example, is that these grammars for HTML, and CSS, and JavaScript are so complicated that it's very easy to confuse the browser's parser.

So for example, let's say that you had something like this. And this is a very common thing to do in frameworks like Django. So you have some div. And then you want to set its class dynamically. So you set its class to some var, so on and so forth. So the idea is that when Django processes this, it should figure out what the current styling is and then put it in here. Well, one thing you can do is maybe the attacker supplies something like a string like this. So attacker will say, class 1. OK, so far so good, because that seems like a valid CSS expression. But then the attacker will then try to put some JavaScript here.

So it might say, onclick equals-- and then put JavaScript URL. And then put some function call here. So this is malformed. The browser should probably just do a fail-stop here. But the problem is that if you've ever looked at the HTML for a real web page, all of it's broken, even for like legitimate, benevolent sites. People just can't hack HTML. So if the browser were to be fail-stop, literally, no site that you enjoy would ever work ever.

If you ever want to be disappointed by the world if I haven't helped you do that enough, open up your JavaScript console when you browse a website and see how many errors get spit out. Like, go to CNN and just see how many errors get spit out. CNN basically kind of works, but it's very disturbing, because if you were to open up Acrobat reader and you're just constantly throwing null pointer exceptions, you would feel a bit cheated by life. But in the web, apparently, we've learned to accept this.

So because browsers have to be so tolerant of these things, they will actually try to massage malformed code into something that seems reasonable. And therein lies a security vulnerability.

So I guess the take-home point for this is that content sanitization kind of works. So it is literally better than nothing. It can actually catch a lot of cases. But in many cases, it is not a full defense.

And so one thing you might actually think about doing is-- actually, let's put this over here. You might think about sort of using a less expressive markup language. So what do I mean by that? So HTML and CSS and JavaScript are [? touring ?] complete. They allow you to do all kinds of fun things, but-- yeah?

AUDIENCE: Sorry to bother you. When does content sanitization not work?

PROFESSOR: When does content--

AUDIENCE: In many cases, it doesn't work.

PROFESSOR: Oh, yeah. So like in this case, for example, Django will probably not be able to statically determine this is a bad thing. Like, in this particular case. But in the case where I inserted that malformed image tag-- I basically said--

AUDIENCE: In that particular case, I would expect the class=assignment to be in quotes and then for that thing to not have any effect. So Django could enforce codes that [INAUDIBLE].

PROFESSOR: Well, see, there's a little bit trickiness there, because if we assumed that all pages were written-- well, pull me back up a little bit. If we assume the HTML grammar was well specified and the CSS grammar was well specified and so on and so forth, then you could imagine a world in which perfect parsers would be able to sort of catch these problems or somehow convert them to normal things. But in fact, the HTML grammars and the CSS grammars are not well specified.

And then on top of that, browsers don't implement specs. So it's like Babushka dolls of terror. So I mean, this, in fact, gets into this notion here. Because I think essentially what you're saying is, well, look, if we have the grammar for something, that should mean something. And as it turns out, if you stick to a less expressive grammar, then it is actually much easier to do content sanitization.

There's some language. It's called Markdown instead of markup. [? Wall, ?] right? And so with Markdown, the basic idea is that it's designed to be a language that allows, for example, users to submit comments, but it doesn't actually have things like the blank tag, and applet support, and stuff like that. And so in Markdown, it's actually much easier to do what you suggested, which seems like a reasonable thing at first glance. Just define the grammar unambiguously and then just enforce that grammar.

So it's much easier to do sanitization in a simple language than in the full-blown HTML, CSS, and JavaScript. And in a certain sense, think about it like the difference between understanding gnarly C code versus gnarly Python code. There's actually a big difference in trying to understand that much more expressive language. Because it can do many more things. By constraining expressivity, you oftentimes improve security. Does that all make sense? All right.

So another thing that you can imagine doing to protect against cross-site scripting attacks is to use something called CSP, Content Security Policy. And so the idea behind CSP is that it's going to allow a web server to-- oh.

AUDIENCE: Yeah, I'm just curious about this Markdown language. So all browsers know how to parse this language?

PROFESSOR: No, no, no. So what happens with a lot of these types of languages is that you essentially-- you can convert them. You can pile them down to HTML, but they're not natively understood by the browser, typically. So in other words, you've got some comment submission system. It internally expresses stuff in Markdown. But then before it can be rendered to the page, it essentially goes to the Markdown compiler. The Markdown compiler then translates it to HTML.

AUDIENCE: I see. Thanks. [INAUDIBLE] Markdown might not be the best trick [? to use Markdown ?] [INAUDIBLE].

PROFESSOR: So Markdown does allow inline HTML. As far as I know, there's a way to disable that in the compiler. I could be wrong about that. But I believe that there's a flag you can pass to get rid of it. But you're correct. If you use a constrained language but then you embed an unconstrained language, then that-- I mean, the terrorists have won. So you're right about that. OK. Yeah.

So another thing you can do to improve security is this thing called Content Security Policy. So like I was saying, what this allows the server to do is to tell a web browser what types of content can be loaded in the page it's sending back, and also where that content should come from. So for example, in an HTTP response, the server might be able to say something like this. It'd include the Content Security Policy header. And then it might say something like the default source is going to equal self. And it will also accept things from asterisk mydomain.com.

So what does this mean? So essentially, the server is saying the content from this site should only come from whatever it is that the domain is for the particular page. And any other subdomain from mydomain.com. So what that means, basically, is that let's say if self was bound to foo.com, let's say, that's the origin of the server that's sending this thing back to the browser.

So if, somehow, there is a cross-site scripting attack and the page tried to generate a reference to, let's say, bar.com, the browser would say, OK, bar.com is not self. Bar.com is also not in this sort of set of domains. So therefore, the browser can just say, I will not allow that request to go forward. So this is actually a pretty powerful mechanism. And you can actually specify more fine-grained controls here. You can say, my images should come from here. My scripts should come from here, so on and so forth. This is actually pretty nice.

And one nice thing about this, too, is that it actually prevents inline JavaScript. So you can't have script tag and then some literal JavaScript and close script tag. Everything has to come from a script tag with a source. So it can be validated through this. And also, a Content Security Policy prevents these danger statements like eval. So eval basically allows a web page to check dynamically generated JavaScript code. And so if the CSP header is specified, the browser does not execute evals. So does that all make sense?

AUDIENCE: So since it's a kind of ad-hoc set of things, is that like a complete set of things that it [INAUDIBLE]?

PROFESSOR: No. So there's a whole list of resources that it actually protects. So this is sort of like the most blanket type protection you could get. But like I said, it actually allows you to specify, I think, like, where CSS can come from, like a bunch of different things.

AUDIENCE: But on preventing evals, that seems like the system's [INAUDIBLE]. Are there are other things [INAUDIBLE]?

PROFESSOR: So yeah, there are. So there's always this question of completeness. So for example, eval is not the only way JavaScript can actually generate code dynamically. There's the function constructor, for example. There's certain ways you can call a set timeout. You pass in a string. You can evaluate code that way. So I believe that CSP actually shuts down those vectors as well. But if you're asking, is this provably complete in terms of what it isolates, no. And I don't think that any of these solutions are provably complete.

AUDIENCE: One really interesting thing about CSP is the fact that you can set it to disallow all inline [? dom ?] script on a page.

PROFESSOR: Yeah. That's right. Yeah, yeah.

AUDIENCE: Which [? helps ?] [INAUDIBLE] to be sanitized.

PROFESSOR: Yeah.

AUDIENCE: [INAUDIBLE] prevents an attacker from--

PROFESSOR: So that helps with some things. But that still would allow, like, [INAUDIBLE] to use eval. So that's why it's important to try to get rid of all of those dynamically. All of those interfaces [? use dynamic ?] code generation.

AUDIENCE: If you list your tag with a source but then also inline code, is there like standardized [INAUDIBLE] that all browsers do with--

PROFESSOR: Yeah. So what should happen is that the inline code should be ignored. The browser should always get the code from the source attribute. I actually don't know if all browsers do that. I've actually personally experienced browsers exhibit different behavior [? in that. ?] This was a couple years ago, so I'm not sure. And so yeah. So one thing to keep in mind about doing work in web security is that in a sense, it's almost like a natural science. So it's like people actually propose theories about how browsers work. And then you go seeing them do that. And so that can be a little bit disappointing, because we're taught, yay, algorithms, and proofs, and stuff like that.

But these browsers are so ill-behaved that a lot of times, the answer is maybe or maybe not. And then [? you go ?] see, as we'll see. They keep on adding features. It gets back to your question about, are these things provably complete? I think web vendors have punted on this notion of creating a browser that is provably [INAUDIBLE]. Basically, what they try to do is just try to keep one step ahead of the attackers. And we'll see some examples of that further in the lecture. So yeah. So CSP is actually pretty cool.

Another thing that's useful is that the server can set this HTTP header called X-Content-Type-Options, and then can say, nosniff. And so what this means is that this prevents the browser from doing some of those, quote, unquote, helpful optimizations, like we discussed last lecture, where it will say, a-ha, there's a mismatch between the file extension and the actual [? bytes ?] that I have sniffed in the contents.

So let me somehow massage this content to some different thing. And then all of a sudden, you've given the barbarians the keys to the kingdom. So you can set this header to basically say, browser, do not do that. And so that can be useful in mitigating some types of attacks as well. All right. So that's kind of a quick survey of some of these cross-site scripting vulnerabilities.

So now let's look at another popular vector for attacks. And that vector is going to be SQL. And so you've probably heard of these SQL injection attacks. And so what these attacks do is they take advantage of the fact that on the back end, for a lot of websites, there's some type of database. And so to dynamically construct a page that's shown to the user, there have to be some database queries that are issued to that back-end server.

So imagine that you have some query that looked like this. So you do a SELECT asterisk. So give me all the values from this query FROM some particular table, WHERE the User ID field is equal to something that is specified over the web from some potentially untrusted source.

So at this point, I may think we all know how this story ends. It ends very badly There are no survivors. So basically, if this comes from someone untrusted, then you can do all kinds of [? chicaner ?] stuff here. So one thing you could do is if you want to be a jerk, you could just set this to the string, 0 and then something like DELETE TABLE. So what happens here?

So basically, the database server's going to say, OK, I'll set the user ID to 0;. Here's a sort of a new command. DELETE TABLE. OK, cheers, there goes your table. And you're done. And in fact, there was a viral image that went around a couple years ago. It's unclear if it was true, like many of these viral images. But it was that people in Germany had license plates that actually said 0; DELETE TABLE.

[LAUGHTER]

Because the idea is that the security cameras, they would use OCR, Optical Character Recognition, to figure out what your license plate was, and then put it in a SQL database. And there were images floating around. These Volkswagens, people would have this as their license plate. I don't know if that works. It's funny. So I like to believe that it's true. But who knows. But you get the basic idea behind that. So once again, the idea is you want to be sure to sanitize this content that you're getting from these untrusted sources.

And so note that there may be some sort of straightforward things that don't quite work. So you might think, OK, well then why can't I just put another quote here and then put another quote here such that whatever it is that the attacker submits, it's going to be enclosed in a string? So this doesn't work, because then the attacker can always just put a quote inside his or her attack string. So a lot of times, these sort of half-hearted hacks don't really get you the security you think they might.

So the solution here is that you need to rigorously encode your data. And once again, that just means that when you get information from an untrusted source, don't just stick it in the system sort of as it is. Make sure that, for example, it can actually escape from whatever sandbox or whatnot you think you're actually putting into. So for example, you want to put in an Escape function that would prevent maybe the semicolon operator from showing up in a raw form and things like this. And so a lot of these web frameworks like Django will actually have built-in libraries to do things like character escaping for SQL queries to try to prevent some of this stuff.

And a lot of these frameworks actually encourage developers not to ever directly interface with the database. So it's like Django itself would provide some high-level interface which does sanitization for you. It takes care of some of these icky corner cases. But performance, performance, performance. Sometimes people think that these web frameworks are too slow. So you will still see, on the back end a lot of time, people will still make these raw SQL queries. And that can lead to problems.

So you can also imagine that there are problems if the web server takes in path names from untrusted images. So imagine that somewhere in your server, you do something like this. You have an open call. And then you say that you're going to read from the WWW directory. You're going to read from the images subdirectory in there. And then you're going to read from some file name that, once again, is supplied by the user.

So as we saw in some of the discussion of [? Troot ?] and things like this, what if this file name maps to something like a bunch of instances of the dot dot character? So if you're not careful, then the untrusted entity can specify basically glub, glub, glub, glub, and go down to etc password and may be able to do some evil here. So once again, if you want to be able to use the web server or the web framework, you need to be able to detect these dangerous characters, escape them in some way to prevent sort of those raw commands from executing. So yeah, it's all pretty straightforward. OK.

So let's move on from the discussion of content sanitization, and now let's talk a little bit about cookies. So cookies are a very popular way to do session management, to bind the user to some set of resources that exist on the server side. And so a lot of frameworks like Django, like [? zoobar ?] that you see in this class, they actually put a random session ID inside the cookie. And so the idea is that this session ID is the index into some server-side table. So you just supply the session ID there. And this is where your user info lives.

And so as a result, this session ID and cookies, by extension, are very sensitive entities. And so that's why a lot of attacks involve stealing of the cookie in order to get that session ID. And so as we discussed in the last lecture, the same origin policy can help you, to a certain extent, against some of these cookie-stealing attacks, because there are origin-based rules that prevent arbitrary tampering with cookies.

But one thing that's a little bit subtle is that you shouldn't share a domain or a subdomain with someone that you don't trust. Because as we discussed in last lecture, there are these sort of very subtle rules in which two origins with the same domain or possibly some subdomain relationship, they can actually access each other's cookies. And so if you trust a domain that you shouldn't, then that domain may be able to do things like directly set the session ID in that cookie that both of you can access. And that can do things like allow the attacker to force the user to use a session ID of the attacker's choosing.

And then, for example-- let's say the attacker sets the user's Gmail cookie, let's say. The user goes to Gmail, types some emails. The attacker, later on, can then use that cookie or specifically use that session ID, load up Gmail, and then access Gmail as if he or she were the user who was victimized. So there's a lot of subtleties with using these cookies for session management. So there's a lot more we could talk about cookies. We'll discuss some of it today and last lecture.

So you might be thinking, well, can we just get rid of cookies? Cookies just seem more trouble than they're worth, just like [? dribbels. ?] So can we just not have these cookies? So one thing you could imagine is you could imagine basically having some notion of stateless cookies, of somehow getting rid of the notion of sessions altogether and preventing this nasty attack vector that seems to be sort of prevalent in all these discussions that we have.

So the basic idea here is if you want to go sort of stateless, then this essentially means you have to authenticate every request. Because the nice thing about cookies is that they basically follow you wherever you go. So you authenticate once, and then every subsequent request you make has this little token in it. But if you want to get rid of those things, well then now you essentially have to have some proof of your authority in every request that you make.

And so one way you could imagine doing this is by using something called MAX, or Message Authentication Codes. And so the basic way to think about one of these MAX, it's like a hash that takes in a key as well. So the method authentication code is the hash of some key and then some message. And so the basic idea is that the client, the user, and the server are going to share some secret key, k. And so the client uses that key to produce a signature over the message that it sends to the server. And then the server, who also knows the key, can then use this same function here to validate if a signature is correct. OK.

So let's look at a very specific example of how this works. So one real service that uses these types of stateless cookies is Amazon Web Services. So like x3, for example. And so basically, Amazon, AWS, gives each user two things-- gives that user a secret key. And so this is equivalent to the k that we were discussing over there. And it also gives them a-- just think of it like an AWS user ID. So this part is not secret, but this part is.

And so every time you want to send a request to AWS via HTTP, you have to send it in a special format. So you'll have the first line of the GET request. So you want to access some photos. No surprises here. And then you will put the host from which you expect to get it. That's not super important. So this is just some AWS server that's there. You'll have the date. So maybe this is Monday, June 4. Whatever.

And then you have this thing that's essentially the Authorization field. And this is where the message authentication code comes in. So essentially, what this looks like is you've got some string here. This represents your access ID, the user ID. And then you've got something here, some other seemingly random letters. And then these things are a signature that use this Message Authentication Code here.

So what does that signature look like? So the details are a little bit complicated. But basically, this signature is over a string that encapsulates a bunch of details of this request. So essentially, the string assigned looks something like this. So you put the HTTP verb in there. So in this case, that verb is GET. And then you put [? indy5 ?] checksum of the message content. And then you also put the content type. So it's html or image or whatever. And put in the date. And then the resource name, which is essentially the path that you see over here.

So in other words, this string here is the message that you pass into the H MAC over here. And so note that the server can see all this stuff in clear text in the request. And so that's what allows the server to validate that that message authentication code was correct. Because note that the server shares that key with the user. So that allows the server to validate that kind of stuff. So does that all make sense?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Oh. So in this case, for the content, that's probably going to be nothing, like the empty string. But you can imagine there's like a post or something like that. You'd actually have the data of the HTTP.

AUDIENCE: [INAUDIBLE] which is kind of an unfortunate choice nowadays.

PROFESSOR: So I believe that they do. So I checked the Amazon documentation yesterday. So I believe they do use it. But I think-- I could be wrong, but I think they actually use a stronger hash here. So that helps a little bit. But you're right. [? Indy5 ?] is not the best.

AUDIENCE: [INAUDIBLE] this works.

PROFESSOR: OK. So allow me to help you, hopefully, even though I'm the guy who confused you in the first place. So the basic idea is that we want to get rid of this notion of this persistent cookie that's always following the user around. Now, the problem, though, is that the server needs some way to identify which client it's talking to. So what we're going to do is we're going to ensure that each client shares a unique key with the server. And so basically, whenever the client sends a message to the server, the client is going to send the message before and then also send this special cryptographic operation, the result of this operation here.

AUDIENCE: Oh, OK. [INAUDIBLE] and then again, you hash it.

PROFESSOR: Yeah. So basically, to first approximation, like imagine in the regular world, like, this would be some cookie here instead of the authorization. But now we're getting rid of the cookie, and we're saying, here's this clear text message. And then here's this crypto thing, which basically allows the server to figure out who this thing came from. And so the server knows who the user is, because that's embedded in the clear. That's not a secret, right? But this basically allows the server to say, aha, I know which secret key this user should have been using to create this if that is, in fact, the real user.

AUDIENCE: Nice. OK. Thanks.

AUDIENCE: So what prevents the attacker from finding the key? Where is this secret key?

PROFESSOR: Yeah, that's a good question. So in a lot of cases, the client for AWS is not a browser, but some VM running in the cloud, for example. So you'll see sort of just VM and VM communication. You can also imagine, too, that users can sort of hand out these links or embed them somehow in HTML. So it's like you just have sort of this-- inside the HTML or JavaScript source code, you'd have the code to create a request like this. So that's almost like me giving you a capability. So if I give you one of these things, you can make that request on my behalf, basically.

AUDIENCE: So would it be possible to use MACs on the normal clients [INAUDIBLE]?

PROFESSOR: For a normal-- you mean like browsers?

AUDIENCE: For normal users.

PROFESSOR: Well, I mean, you get into these questions like, where does the key live, which was [? kind of like what ?] he was asking. So in a certain extent, the issue of where the key lives is actually super, super important. Because if the key can be stolen just as easily as the cookie, well then we're sort of back to square one. So in many cases, this stuff is actually just, as I said, sort of server to server, like a VM to VM somewhere in the cloud. So the application developer runs a VM that sort of outsources a bunch of stored stuff to AWS.

AUDIENCE: So do you think [INAUDIBLE] but isn't that kind of like a bad way of preventing-- I mean, they have network latency, so it can't be like too fine-grained of a constraint that they're putting on. If an attacker sends the same request again really quickly after the user, wouldn't they be able to [INAUDIBLE]?

PROFESSOR: Yeah, yeah, yeah. So suffice it to say that secure timestamping is like several people's PhD theses. But you're exactly right that if this-- just as a crude example. So imagine this just said, Monday, June 4. Then if, somehow, the attacker could get access to this entire thing and there was nothing that was different about it-- so there was no [? knocks, ?] no random stuff like that, then that's right. Then that request could be [INAUDIBLE].

Now, one thing that AWS actually does is you can actually include an expiration date in these things. So one thing you can actually do is add sort of an Expires field, essentially, have that thing be assigned. Then I can hand that reference to a bunch of different people. Kind of like I was saying in response to his question, it acts as a capability. The server can then check that expiration date from when it actually sees it and then not actually--

AUDIENCE: But even if the expiration date is like 200 milliseconds in the future or something, as long as the attacker has [INAUDIBLE] latency, then they might send two [INAUDIBLE] two copies instead of one.

PROFESSOR: Yeah. That's exactly right. That's exactly right. So yeah, if the attacker can somehow-- like a network attacker, for example, is seeing these things go over the wire-- and you're right. If there's enough wiggle room in the expiration date, then they can exactly do that attack. That's right. OK. So that is an overview of how these stateless cookies work.

And so one question that's interesting is you might think, well, what does it mean to log out with this type of cookie? And the answer is that you don't really log out. I mean, you have this key. And so whenever you want to send a request, you just send it. You include this dude right here, and then you're ready to go.

Now, one thing the server could do, for example, though, is revoke your key. So the server revokes your key. Then you can generate one of these things. But when you send the message over there, the server's going to say, aha, I know what your user ID is. You've been revoked, so I'm not going to honor your request. But it's a little bit interesting. And revocation, as we'll talk more about with things like SSL, is always a tricky issue, because as it turns out, taking authority away from human users is often much more difficult than giving it to them in the first place. So that's the basic idea behind these sort of stateless cookies.

So there's also a couple other things that you can use if you want to avoid traditional cookies for implementing authentication. So one thing you can imagine doing is actually using DOM storage to hold client-side authentication information. This says "alternatives" in case you couldn't [? read that. ?]

So one thing you could do is to use DOM storage to hold some of that session state that you would ordinarily put inside of a cookie. So if you remember from last lecture, DOM storage is essentially a key value interface that the browser provides to each origin. So you can say GET and PUT in both the key and eval [? strings. ?] So you could imagine putting authentication stuff inside there. Now, the nice thing about this is that DOM storage actually has much less wacky rules with respect to the same origin policy. So if it were cookies, you can do all these tricks with subdomains and stuff like that. It got kind of weird.

DOM storage is actually strictly tied to a single origin. You can't do any of this subdomain expansion, all that kind of stuff. And so frameworks like Meteor use DOM storage for this very reason. But now, note that if you want to store authentication information in DOM storage, then you have to write JavaScript code yourself to actually pass that authentication information to the server to do the [? encryption ?] that's necessary and so on and so forth. So that's one thing you could do.

Another thing you could do is actually use client-side certificates. So for example, like an x.509 format. And so what's nice about these certificates is that, basically, JavaScript has no explicit interface to access these things. So unlike cookies, where there's always this arms race to find these weird same-origin bugs, there's no explicit JavaScript interface for that stuff. So that's very nice from a security perspective.

One problem that I mentioned very briefly that we'll look at in more detail in later lectures is that the revocation [? store ?] is kind of hard for these. So once a user leaves your organization, how do you take back their certificates? And it becomes a little bit tricky.

Also, these things don't have great usability, because who wants to install a bunch of certificates for each site that you go to? So as a result, these things have a lot of friction, and these are not super popular except for in companies or organizations that are super security-conscious. All right. So that concludes our discussion of cookies.

And so now let's talk about protocol vulnerabilities in the web stack. And so one kind of interesting attack is that there are all these bugs in the way that different browser components parse URLs, for example. So how can URL parsing get us into trouble? So suppose that we have a URL that looks like this. HTTP comes from example.com. And then it's got an explicit port specifies that it's 80. And then for some unknown reason, it embeds this @ character here.

So the question is, well, what is the origin of this particular URL? So as it turns out, so Flash would say that the host name portion of this was example.com. However, when the browser would parse this, it would say that the host name part of the origin was actually foo.com. So this is clearly a bad thing, because once you have two different entities who are confused about the origin of the same resource, then you can get into all kinds of nasty problems.

So for example, the Flash code can be malicious, can download stuff from example.com. If it was embedded in the page from foo.com, it could then do some evil things there. And it takes some code from example.com and run it with the authority of foo.com. So there's a lot of complex parsing rules like that that make life very difficult. This is a continuing theme. Like, as we just saw with the content sanitization-- so the basic idea that it's oftentimes much better to have simpler parsing rules for this kind of stuff. It's difficult to do that in retrospect, though, because HTML's already out there. So all aboard the wam-bulance.

So this next one, this is actually my all-time favorite security vulnerability. So it basically attacks the way that the browser [? rule 1 ?] JAR files, basically Java applets. So in 2007, I think-- yeah, 2007. So lifehacker.com-- great website if you haven't been to it. Lifehacker.com basically explains how you can embed ZIP files inside of images.

Now, it's not quite clear who you're trying to hide from by doing this. But Lifehacker says you can do it, so hurray. So basically, what they take advantage of is the fact that if you look at image formats like GIF, for example, typically, the way the parser works is the parser works from the top, down. So it finds information in the header. And then it sort of computes on the rest of the bits here.

Now, what was interesting is that, as it turns out, programs which manipulate ZIP files typically work from the bottom up. So they find some information in the footer of the file. Then they work up to try to extract what's inside of it. So what Lifehacker basically said is that if you wanted to hide a ZIP file on a merger or something like this, then you could actually post a GIF there that has this ZIP file here.

It will pass all the validation checks on Flickr or whatever as an image. It will actually display as an image in your browser. Aha, but only you know the hidden truth, that if you take this file here, you can pass it to unzip, and it will unzip [INAUDIBLE] information there. OK, fine, this seems like it's sort of like a cheap parlor trick. OK, that's nice.

Now, attackers, of course, never sleep, and they want to ruin our life. So what did they realize? They realize that JAR files are basically derivatives of the .ZIP format. So this meant that you could actually create a GIF or an image that had a JAR file, executable JavaScript code, at the bottom of it. So then people called this attack-- they called it the GIFAR attack.

[LAUGHTER]

Half GIF, half JAR, all evil. Because this was amazing. And so what did this mean that you could do? Well, it's actually quite subtle. Because people first discovered this, they thought it was amazing, but they didn't quite know how to exploit it. But as it turns out, you can do things like the following. So first of all, how do you make one of these things? You just use CAD. There is literally no [? trickeration ?] that you have to do. Take this, take this, you CAD it. Boom, you've got a GIF/JAR.

So once you have that, what can you do? Well, there are some sensitive sites that will allow users to submit data, but not arbitrary types of data. So [INAUDIBLE] Flickr or something like that, it may not allow you to submit arbitrary ActiveX or whatever, arbitrary HTML. But it will allow you to submit images. So what you could do is construct one of these things, submit it to one of these sensitive sites that does allow you to submit images. And then what can you do? Well, the next thing you need to do is-- so yes, the first thing you do is you submit one of these things to the sensitive [? cycle. ?]

And then the next thing that you can do is if you have an XSS attack, if you have a cross-site vulnerability, then you can use the cross-site scripting to inject something like this. And due to poor board management, I will draw this over here. So you can inject an applet, write JavaScript code that has, as its sort of source, you just say, cats.gif.

And so what's interesting about this is that this code, because we're using a cross-site scripting vulnerability, runs in the context of the vulnerable site. This has been uploaded to the vulnerable site's origin. So this will pass the same origin test. But however, this code was specified by the attacker.

So now what happens is that the attacker gets to run that Java applet in the context of the victim's site with all the authority of that origin even though the GIFAR passed the vulnerable site's image validation code. Because one of these things will actually parse correctly as a GIF. But it has this hidden code in here. And so [INAUDIBLE] when the browser tries to execute the JAR part of it, once again, it starts from the bottom, comes up here, and just ignores that part. So this is actually pretty amazing.

And so there's some fairly straightforward ways you can fix something like this. So for example, you can actually have the applet loader actually understand that there should not be random junk up here, for example. What was happening in many cases is that there was information in the metadata saying, here's the length of this resource. And then if it said, the length, it stops here, they would just say, who cares what's the rest. It's probably zero. But in this case, it wasn't.

What I love about this is that it really shows how wide the software stack is for the web. So sort of taking these two formats, GIF and then JAR, we can actually create this really nasty attack. You can actually do this for PDFs, too. You can put PDFs here. I think that was called, like, the [? PDFR ?] attack or something like this. So people had a field day with this for a day. These vulnerabilities have been closed now.

AUDIENCE: So what can you do with this attack that you can't do with [INAUDIBLE] XSS or your own [INAUDIBLE]?

PROFESSOR: So what's nice-- yeah, yeah. So good question. So what's nice about this is that Java oftentimes can be more powerful than just running regular JavaScript, because it has slightly different rules on, [? same origin ?] policy and stuff like that. [INAUDIBLE] get more lower-level access to the file systems or things like that.

But you're right, that if you can do cross-site scripting, running JavaScript's already pretty damaging. But the main advantage of this is, once again, running inside the applet. All right. Yeah. So like I said, that's my favorite attack of all time, mainly just because it forced serious-minded security individuals to say GIFAR all the time. So if you're easily amused, like myself, then this was a bonanza for you.

So another thing that's interesting is that there are actually attacks that are based on a time.

So you might not think of time as a resource which could be a vector for attacks. But as I was discussing with someone a few minutes ago, yeah, time can actually be a way that a system can be exploited.

And so these attacks are called-- the particular attack I'm going to talk to you about is a specific example of a covert channel attack. And so the idea behind the covert channel attack is that, essentially, the attacker has found some way for two applications to exchange information. And that exchange vector is not an officially sanctioned vector. The attacker is somehow leveraging some other part of the system to pass bits of information between two different entities.

So a good example of some of this stuff is something called CSS-based sniffing attacks. So what is this attack all about? So attacker has a website that the user can visit. And once again, getting a user to visit a website is actually usually pretty straightforward. You create ads. You send them a phishing email, whatever.

So the attacker has a website that the user visits. And the goal of the attacker is to learn what other websites the user has visited. And the attacker might want to know this for several reasons. Maybe they're trying to figure out what kinds of search terms the user's looking for. Maybe they're trying to figure out where that person's employed, or maybe they want to know if they've accessed some type of embarrassing material, so on and so forth.

So how is the attacker going to do that if the only thing that the attacker controls is a website that he or she can convince the user to visit? Well, the exploit is to leverage link colors. So you know like when you go to a web page and you click on a link, the next time you see that link, it is now a different color. So zoinks, that's actually a security vulnerability.

Because what that means is that in this attacker website, if the attacker can trick you into visiting it, then the attacker can generate a huge list of candidate URLs that you might have visited and then use JavaScript to see what color those URLs are. And if the URL color is purple, that means, aha, you have visited that site. So this was very subtle.

And what's interesting about this is that you don't even have to display the URLs in many cases to the user. You can just sort of conjure up a domino that has a particular href and just look at its style, and then see if it has the visited color or not. So this is actually pretty subtle.

So you might be thinking, well, isn't it going to be inefficient to scan through all these candidate

URLs? We can do all kinds of clever optimizations. So for example, you can have multiple passes. In your first pass, you could only see if the user had visited top-level URLs-- cnn.com, Facebook.com, so on and so forth. If the answer is yes, you can then do sort of a depth-first search on those hits that you found at the top level. So you can actually really constrain the search space this way.

So this was really, really funny, too, if you have a demented sense of humor, because it showed that this very innocuous feature that browsers support-- they're just trying to help you out. They're trying to say, hey, buddy, here's where you visited. It can actually reveal this very damaging information.

So what is a solution for this? So in practice, what the browser [? runners ?] did is that they made it such that the browser lies to JavaScript about the color of links. So basically, when JavaScript tries to look at the link and look at its styling, the browser always says, unvisited. OK. So that seems somewhat unfortunate, but it prevents this attack. So I guess we can live with it. JavaScript not being able to read link colors, eh, not the end of the world. So are we done, though? Does this fix the problem of the attacker being able to figure out where you've been? The answer, of course, is no.

So the next attack that the attacker can do is a cache-based attack. And so the intuition here is that, once again, the goals are the same. Attacker wants to know what sites you visited. The exploit vector is that information that has been cached is quicker to access. That, in fact, is the whole reason why you cache it in the first place.

So once again, the attacker can generate a list of candidate objects that the attacker thinks you might have visited and then just time how quickly those objects come back to the attacker. And so if the objects come back quickly, you know [? you need some ?] threshold, the attacker can guess that you, in fact, have been to those objects before. So does that make sense? Once again, the browser's just trying to help you out. But you can leverage these techniques to figure out some evil knowledge.

And what's interesting about this is that this attack can actually leverage some very interesting geographic location information. So imagine that we're doing attacks on Google Map tiles, for example. So if I detect that you've actually accessed a series of Google Map tiles, that probably means you are either in that place or you're interested in other people who might be in that place. So it's actually a pretty powerful attack. So OK.

So how can you fix this one? Well, this one is not quite clear. You could have a site that doesn't cache anything at all. And then your site's going to be slow. So that kind of sucks. So it's not quite clear how you get around this. But OK.

Let's suppose that we have the defense we put in place here-- JavaScript can't read link colors. Let's assume that the site is super paranoid it caches nothing. So have we completely defended ourselves against this attack? One second. So the answer is no. Because the attacker can actually launch DNS-based attacks. So the intuition is that even if you don't cache anything, when you access a resource for the first time, you have to generate a DNS request for the hosting that's associated with that resource.

So once again, the attacker can look in time and see how long it takes for the attacker to access these candidate objects the attacker thinks you may have accessed. And if they come back quickly, then that's perhaps a good hint that you've resulted the DNS name for that host before. And so this works even if you don't cache anything, because the DNS cache lives with the OS, not with the browser.

AUDIENCE: You've mentioned, I think last class, the ability to get JavaScript to take screenshots.

PROFESSOR: Yeah, yeah.

AUDIENCE: So can you just render the [? link ?] as a single pixel, and then take a screenshot, and [INAUDIBLE] that pixel?

PROFESSOR: Yeah. Well-- so you could. So rendering stuff is always a little bit tricky, because you have to play these games. If you want to show something to the user, it has to flash really quickly. Or else they might see that someone's entering this huge list of URLs. But you're right. If you have access to the screen-sharing API, a lot of this becomes a lot simpler.

AUDIENCE: And if you just have some kind of animated image that looks mostly random, then you just pay attention to one pixel of it?

PROFESSOR: You're exactly right. I mean, in general, I think the screen-sharing API is a bad idea. I'm not the president of the world, so what can I do? So anyways, so DNS-based attacks work even if there is no caching that takes place. OK.

So as the final piece de resistance, so you might think, OK, what if we only use raw IP addresses for all of our host names? We don't cache a thing! OK? And we're running on an updated browser that doesn't expose link colors to JavaScript. So surely we're fine. I'm here to tell you you are not fine. Because what the attacker can actually do is take advantage of rendering attacks.

So the basic idea here is that it is typically faster to render a URL that you have visited before for various wacky reasons that have to deal with how browsers [INAUDIBLE] rendering [INAUDIBLE] internal. And so what the attacker can do is actually create a candidate iframe, let's say, puts some content in there that the attacker thinks you may have visited, and then constantly see if the attacker loses access to that iframe. Because as that iframe is loading, the browser typically thinks that iframe belongs to the attacker's page. And then as soon as that different origin content comes in, then you'll start getting these access errors.

Because now that different origin [INAUDIBLE]. So now the attacker can't touch anymore. So the attacker can do things like this still to see if there's caching, rendering information [INAUDIBLE] browser for these candidate sites. So anyways, so those are the only hopes and dreams I want to crush in you today. I believe we're running out of time. But I will see you next time.