PROFESSOR: All right, guys. Let's get started with the next installment of our exciting journey into computer security. Today, we're actually going to talk about web security. Web security is, actually, one of my favorite topics to talk about because it really exposes you to the true horrors of the world. It's very easy to think, as a student, that everything will be great when you graduate.

Today's lecture and the next lecture will be telling you that's, in fact, not the case. Everything's terrible. So what Is the web? Well back in the olden days, the web was, actually, much simpler than it is today, right. So clients, which is to say the browsers, couldn't really do anything with respect of displaying rigid or active content. Basically they could just get static images, static text, and that was about it.

Now the server side was a little more interesting because even if there was static content on a clients side. Maybe the server was talking databases, maybe it was talking to other machines on the server side. Things like that. So for a very long time, the notion of web security, basically, meant looking at what the server was doing. And to this point in this class, we've essentially taken that approach.

So we looked at things like buffer to overflow attacks. So how clients can trick the server into doing things the server doesn't want to do. You also looked at the OKWS server and looked at how we can do some privilege isolation there.

So to this point, we, sort of, looked at security through the experiences that were actually experienced by the security resources themselves. But now, actually, the browser is very interesting to think about, in terms of security, because the browser is super, super complicated these days. So now there's all kinds of insane, dynamic stuff that the browser can actually do.

So for example, you probably heard of JavaScript. So JavaScript now allows pages to execute client side code, Turing complete, can do all kinds of wacky stuff. There is the DOM model, which we'll talk about in more depth later today. The DOM model, essentially, allows

JavaScript code to dynamically change the visual appearance of the page. Fiddle with things like font stylings and stuff like that.

There's XML HTTP request. These are, basically, a way for JavaScript to asynchronously fetch contents from servers. You may also hear XML HTTP requests referred to as AJAX. Asynchronous JavaScript fetching.

There are things like web sockets. This is, actually, recently introduced API. So WebSockets, essentially, allow a full duplex communication between clients and servers. Communication going both ways. We've got all kinds of multimedia support.

So for example, we have things like the video tag, which allows a web page to play video without using a Flash app. It can actually just play that video natively. There's also a geolocation. So now a web page can actually determine, physically, where you are.

For example, if you're running a web page on a smartphone, the browser can actually access your GPS unit. If you're accessing a page on a desktop browser, it can actually look at your Wi-Fi connection and connect to Google's Wi-Fi geolocation service to figure out where exactly you are. That's, kind of, insane. Right?

But now web pages can do do that kind of stuff. So we've also talked about things like NaCl, for example, which allows browsers to run native code. So there's many, many other features that I haven't mentioned here. But suffice it to say the browser is now incredibly complicated.

So what does this mean from the perspective of security? Well basically, it means that we're screwed. Right? The thread surface for that right there is enormous. And loosely speaking, when you're thinking about security, you can think of a graph that, sort of, looks like this.

So you've got the likelihood of correctness. And then, you've got the number of features that you have. And so you know, this graph starts up here at 100. Well of course, we never even started 100, even with very simple code because we can't even do bubble sort right.

So essentially, that curve looks something like this. And web browsers are right over here. So as we'll discuss today, There's all kinds of wacky security bugs that are arising constantly. And as soon as the old ones are fixed, new ones are rising because people keep adding these new features. Oftentimes, without thinking about what the security implications of those features are.

So if you think about what a web application is these days, well it's this client thing and it's a server thing. And a web application now spans multiple programming languages, multiple machines, and multiple hardware programs. You could be using Firefox on Windows.

Then it's going to go talk to a machine in the cloud that's running Linux. It's running the Apache server. Maybe it's running an ARM chip opposed to x86 or something like that, or the other way around. So long story short, there's all these problems of composition. There's all these software layers and all these hardware layers that all can impact security in some way.

But it's also complicated. It's not quite clear how we can make sense of the entire whole. So for example, one common problem with the web is this problem of a parsing context. So as an example, suppose that you had something in a page that looked like this.

You declare a script tag. Inside that script tag, you declare a variable. There's some string here. And let's say that this string comes from an untrusted party. Either the user or another machine or something like that.

And then, you close that script tag. So this stuff is trusted. This stuff is trusted. This stuff is not trusted. So can anybody figure out why there might be some problems here if we take this entrusted string and put it in there?

AUDIENCE: You can have a closing quote mark in [INAUDIBLE] and then have some [INAUDIBLE].

PROFESSOR: Right, right, exactly. So the problem is there are multiple context, that this untrusted code could, sort of, break into. So for example, if the untrusted code had a double quote here, now we've closed the definition of this JavaScript string.

So now we're added the JavaScript string context and render the regular JavaScript execution context. And then the attacker gets a regular job zip code here and go to town. Alternatively, the attacker could just put a closing script tag here. Right?

And then, at that point, the attacker can, sort of, get out of the JavaScript context and then get into the HTML context. Maybe to find some new HTML nodes or something like that. So you see this problem with composition all over the place in the web because there are so many different languages and run times for you to think about.

HTML, CSS, JavaScript, maybe MySQL on the server side, and so on and so forth. So this is just a classic example of why you have to do something called content standardization. So

whenever you get untrusted input from someone, you actually need to analyze it very carefully to make sure that it's not being used as a vector for an attack.

So another reason why web security so tricky is because the web specifications are incredibly long, they're incredibly tedious, they're incredibly boring, and they're often inconsistent. So when I mean the web specifications, I mean things like the definition of JPEG, the definition of CSS, the definition of HTML. These documents are, like, the size of the EU constitution and equally as easy to understand.

So what ends up happening is that the browser vendors see all these specs. And they essentially say, OK, thanks for that. I'm going to do something that somewhat resembles what these specs look like. Then they call it a day and they laugh about it with their friends.

OK, so what ends up happening is that these specifications end up being like these vague, aspirational documents that don't always accurately reflect what real browsers are doing. And if you want to understand the horror of this, you can go to this site called quirksmode.org. I mean, don't go to this site if you want to be happy.

But you can go there. And it actually documents all of these terrible inconsistencies that browsers have with respect to what happens when the user hits a key press? There should just be one key precedent that's generated. You are so wrong. So go to quirksmode.org and check that out, and see what's going on.

So anyway, in this lecture, we're going to focus on the client side of the web application. In particular, we're going to look at how we can isolate content from different web providers that has to coexist, somehow, in the same machine and the same browser. So at a high level, there's this fundamental difference between the way you traditionally think of a desktop application and the way you think of a web application.

Abstractly speaking, most of the desktop applications that you use, you can think of it as coming from a single principal. So word comes from Microsoft. And maybe TurboTax comes from Mr. and Mrs. TurboTax, so on and so forth. But when you look at a web application, something that looks to you, visually, as a single application is actually composed of a bunch of different content from a bunch of different people.

So you go to CNN, it looks like it's all on one tab. But each of those visual things that you see may, in fact, come from someone different. So let's just look at a very simple example here. So

let's say that we were looking at the following site. So HTTP food.com. And we're just looking at index.html.

So you know, you look at your browser tab. What might you see? So one thing that you might see is an advertisement. So you might see an advertisement in the form of a gift. And maybe that was downloaded from ads.com.

Then you also might see, let's say, an analytics library. And maybe this comes from google.com. So these libraries are very popular for doing things like tracking how many people have loaded your page, looking to see where people click on things to see which parts of their site are the most interesting for people to interact with, so on and so forth.

And you might also have another JavaScript library. Let's say it's jQuery. And maybe that comes from cdn.foo.com. So some content distribution network that foo.com runs. jQuery is very popular library for doing things like GUI manipulation. Things like that. So a lot of popular websites have jQuery. Although, they serve it from different places.

And then, on this page you might see some HTML. And here's where you might see stuff like buttons for the user to click on, text input, and so on and so forth. So that's just raw HTML on the page.

And then, you might see what they call inline JavaScript from foo.com. In my inline, you have a script tag. And then, you have a closed script tag. And then you just have some JavaScript code included in their directly.

That's as opposed to where you say something like script. And then, the source equals something that lives on some server remotely. So this is what's called inline JavaScript. This is what's referred to as an externally defined JavaScript file. So you might have some inline JavaScript there from foo.com.

And the other thing that you might have in here is actually a frame. So we'll talk about frames a bit more in a little bit, but think of a frame as almost like a separate JavaScript universe. It's a little bit equivalent to a process and UNIX.

So maybe this frame here, maybe this guy belongs to https://facebook.com/likethis.html. So maybe here we have some inline JavaScript from Facebook. And then, maybe, we also have some image.

So you know, f.jpeg. That comes from https://facebook.com. OK, so this is what a single tab might have in its contents. But as I just mentioned, all this can, potentially, come from all these different principles. So there's a bunch of interesting questions that we can ask about a application that looks like this

So for example, can this analytics code from google.com actually access JavaScript state that resides in the jQuery code. So to first approximation, maybe that seems like a bad idea because these two pieces of code came from different places. But then again, maybe it's actually OK because, presumably, foo.com brought both of these libraries in so that they can work with each other. So who knows.

Another question you might have is can the analytics code here actually interact with the text inputs here. So for example, can the analytics code define event handlers? So a little bit of background in JavaScript. JavaScript is single threaded vent driven model.

So basically, in each frame, there's just an event loop that's just constantly pulling events. Key presses, network events timers, and stuff like that. And then, seeing if there are any handlers associated with those events. And if so, it fires them. So who should be able to define event handlers for this HTML. Should google.com be able to do it. It's not from foo.com so maybe, maybe not.

Another question, too, is what's the relationship between this Facebook frame here and the larger frame? The Facebook frame is an HTTPS, secure. foo.com is an HTTP, nonsecure. So how should these two things be able to interact?

So basically, to answer these questions, browsers use a security model called the same-origin policy. So there's, sort of, this vague goal because a lot of things with respect to web security are, kind of, vague because nobody knows what they're doing. But the basic idea is two websites should not be able to tamper with each other, unless they want to.

So defining what tampering means was actually easier when the web was simpler. But as we keep adding these new APIs, it's more and more difficult to understand what this non-tampering goal means. So for example, it's obviously bad if two websites, which don't trust each other, can over write o each other's visual display. That seems like an obviously bad thing.

It seems like an obviously good thing if two websites, which want to collaborate, are able to,

somehow, exchange data in a safe way. So you can think of mash up sites you may have heard of. So sometimes you'll see these things in the internet.

It's like someone takes Google map data, and then takes the location of food trucks. And then, you have this amazing mash up that allows you to eat cheaply and avoid salmonella, right? So that seems like a thing you should be able to do. But how, exactly, do we enable that type of composition?

Then there's other things that are, kind of, hard to say. So for example, if JavaScript code comes from origin x inside of a page that's from origin y, how exactly should that code and that content compose? So the strategy that the same-origin policy user can be roughly described as follows.

So each resource is assigned an origin, which we'll discuss in a second. And essentially, a JavaScript code can only access resources from its own origin. So this is the high level strategy the same origin policy uses.

But the devil's in the details. And there's the ton of exceptions, which we're going to look into in a second. But first of all, before we proceed, let's define what an origin is. So an origin is, basically, a network protocol scheme plus a host name plus a port.

So for example, we can have something like HTTP foo.com. And then, maybe, it's index.html. So the scheme here is HTTP. And the host name is foo.com. And the port is 80.

Now the port, in this case, is implicit. The port is the port on the server side that the client uses to connect. So if you see a URL from the HTTP scheme and there's no port that's explicitly supplied, then, implicitly, that port is 80.

So then, if we look at something like the HTTPS, once again, foo.com index.html. So these two URLs have the same host name. Right? But they have, actually, different schemes. HTTPS vs HTTP.

And also, here, the port is implicitly 443. That's the default HTTPS port. So these two URLs have different origins.

And then, as a final example, if you had a site like HTTP bar.com, then you can use this colon notation here. 8181. You know, these things beyond here don't matter with respect to the same origin policy, at least with respect to this very simple example.

Here, we see that we have a scheme of HTTP, a host name of bar.com, and here we've explicitly specified the port. So in this case, it's a non-default port of 8181. So does that make sense? It's pretty straightforward. OK, so this is, basically, what an origin is. Loosely speaking, you can think of an origin as a UID in Unix with the frame being loosely considered as, like, a process.

So there are four basic ideas behind the browser's implementation of the same origin policy. So first idea is each origin has client side resources. So what are examples of those resources? Things like cookies.

Now you can think of cookies as a very simple way to implement state in a stateless protocol like HTTP. Basically, a cookie is like a tiny file that's associated with each origin. And we'll talk about the specifics of this in a bit.

But the basic idea is that when the browser sends a request to a particular website, it includes any cookies that the client has for that website. And you can use these cookies for things like implementing password remembering. Maybe if you were going to an ecommerce site, you can remember stuff about a user's shopping cart in these cookies, so on and so forth.

So cookies are one thing that each origin can be associated with. Also, you can think of DOM storage as another one of these resources. This is a fairly new interface. But think of DOM storage as just a key value store.

So DOM storage allows an origin to say, for this given key, which is a string, let me associate it with this given value, which is also a string. Another thing that is social with an origin is a JavaScript name space. So that JavaScript name space defines what functions and what interfaces are available to the origin.

Some of those interfaces are built in. Like, let's say, the string prototype and stuff like that. And then, an application might actually fill the JavaScript namespace with some other content. There's also this thing called the DOM tree.

So DOM is short for Document Object Model. And the Dom tree is, essentially, a JavaScript reflection of the HTML in a page. So you can imagine that the DOM tree has a node for the topmost HTML5 node in the HTML. And then, it's going to have a node for the head tag.

Then, it's going to have a node for the body tag. All right, so on and so forth. So the way that a

lot of dynamic web pages are made dynamic is the JavaScript code can access this data structure in JavaScript that mirrors the HTML content.

So you can imagine an animation takes place by changing some of these nodes down here to implement different organizations of various tabs. So that's what the DOM tree is. There's also a visual display area. Although, we'll see that the visual display area actually interacts very strangely with the same origin policy. So on and so forth.

So at high level, each origin has access to some set of client side resources of these types. Doe that make sense? And then, the second big idea is that each frame gets the origin of its URL. So as I mentioned before, a frame is, roughly, analogous to a process in Unix. It's, kind of, like a name space that aggregates a bunch of other different resources.

So third idea is that scripts, so JavaScript code, execute with the authority of it's frame's origin. OK, so what that means is that foo.com imports a JavaScript file from bar.com. Well, that JavaScript file is going to be able to act with the authority of foo.com.

So loosely speaking, this is, sort of, similar to if you were in the Unix world to run a binary that, sort of, belonged in someone else's home directory. That thing would sort of, execute, with your privileged there.

And the fourth thing is there's passive content. So by passive content I mean things like that images, for example. Or CSS file or things like that. These are things, which we don't think of as having executable code.

So passive content gets zero authority from the browser. So that, kind of, makes sense. We'll see why this fourth thing is a little bit subtle in a second. So going back to our example here. So we see, for example, that the Google Analytics script and the jQuery script can access all kinds of stuff in foo.com.

So for example, they can read and write cookies. They can do things like attach event handlers to buttons here. So on and so forth. If we look at the Facebook frame and its relationship to the larger foo.com frame, then we see that they're from different origins because they have different schemes here. They have different host names. Different ports.

So what this means is that they are, to a first approximation, isolated. Now they can communicate if they both opt into it using this interface called postMessage. So postMessage allows two different frames to exchange asynchronous immutable messages with each other.

So think of this facility as allowing Facebook to try to send a string. Not a reference, a string up to the enclosing foo.com frame. Now note that if foo.com doesn't want to receive those messages, it doesn't have to. So this has to be opt in from both sides to get this thing to work. So note that the JavaScript code here in the Facebook frame cannot issue an XML HTTP request to the foo.com server. That's once again because network destinations also have these origins that are associated with them.

So because Facebook.com does not have the same origin as foo.com it can't asynchronously fetch stuff from it via HTML request. So the last thing we can look at we can say, OK, we got an image up here from ads.com. This is rule number four over there.

So it seems pretty straightforward. This is an image. It has no executable code. So clearly, the browser's going to give it no authority. Now that seems kind of like a dumb thing. Like, why are you even talking about images having authority or not having authority? It seems obvious that images shouldn't be able to do stuff.

Well it's a security class. So clearly, there is mischief that hides in statement number four up there. So what happens if the browser incorrectly parses an object and misattributes it's type?

So you can actually get into security problems there. And this was actually a real security problem. So there's this thing called the MIME sniffing attack. So the MIME type-- I mean, you've probably seen these before. You knows it's Something like text dot HTML or image.JPEG Things like that. This was like a MIME type. So old versions of i.e used to do something that they thought was going to be helpful for you.

So sometimes what web servers will do is they will misattribute the file extension of an object. So you can imagine that a web server that's been configured incorrectly might attach a dot HTML suffix to something that's really an image. Or it might attach a dot JPEG suffix to something that's really HTML.

So what IE would do back in the olden days is try to help you out. So IE would go out. It would go fetch this resource. And it would say, OK, this resource claims to be of some type, according to its file name extension. But then it would actually look at the first 256 bytes of what was in that object.

And if it found certain magic values in there that indicated that there was a different type for

that object, it would just say, hey, I found something cool here. The web server misidentified the object. Let me just treat the object like it's type that I found in these first 256 bytes.

And then, everybody's a winner because I've helped the web server developer out because now their website's going to render properly. And the user's going to like this because they get to unlock this content that would have been garbage before. But this is clearly a vulnerability because suppose that a page includes some passive content.

Like, let's say, an image from a domain that's controlled by the attacker. Now from the perspective of the victim page, it's saying, even if this attacker site is evil, it's passive content. It can't do anything. Like, at worst, it displays an unfortunate image. But it can't actually access any code because passive content gives 0 authority.

But what would happen is that IE could sniff this image. The first 256 bytes. And the attacker could intentionally put HTML and JavaScript in there. So what would happen is that the victim site brings in what it thinks is an image. IE coerces it into HTML and JavaScript. And then, executes that code in the context of that enclosing page. So does that attack make sense? so

This is, sort of, an example of how complex browsers are and how adding even a very well intentioned feature can cause these very subtle security bugs. So let's now dig down and take a deeper look at how the browser secures various resources.

So let's look at frames and window objects. So frames represent these separate JavaScript universes that we discussed over here. I mean, implementation wise, a frame with respect to JavaScript is an instance of a DOM node. So I forget where I drew-- oh, yeah. This DOM node up here.

So the frame would exist as a DOM node object somewhere in this hierarchy that's visible to JavaScript. In JavaScript, the window object is actually an alias for the global name space. It's, kind of, this wacky idea. Like, if you were to find this global variable name x, you can also access it via the name window.x.

OK, so basically, frames and window objects are very powerful references for you to be able to access. And they actually contain pointers to each other. The frame can [INAUDIBLE] a pointer to the associated window object and vice versa.

So these two things are, essentially, equivalently powerful. So frame and window objects get the origin of the framed URL. Or because there's always an or in web security, they can get a

suffix of the original domain name. The original origin.

So for example, a frame could start off having an initial origin. x dot y dot z dot com. So let's ignore the scheme and the protocol for a second. So initially, the page can start off like this. It can then intentionally say I want to set my origin to be y dot z dot com. A suffix of that.

And the way that it indicates this is by doing an assignment to the special document dot domain value that's accessible via JavaScript. So we can set document dot domain explicitly to this right here. And that's allowable because this guy is a suffix of that guy.

And then, similarly, it could also set document dot domain to z.com and effectively reset it's origin like that. Now what it cannot do is it cannot do something like setting document domain to a dot y dot z dot com.

That's disallowed because this is not a problem this is not a proper suffix of the original origin. And also, it cannot set its suffix to dot com. So does anyone have any theories about why this is a bad idea? Right, exactly. So people are laughing because, clearly, this is going to bring out the apocalypse, right.

So if it does this, then this means that the site could somehow be able to impact cookies or things like that in any dot com site, which will be pretty devastating. The motivation for why these types of things are allowable is because, presumably, these origins have some type of preexisting trust relationship. So this seems to be vaguely OK. Whereas, this would seem to be bad.

AUDIENCE: So you can make these splits on any dot or actual end point? Like, for example, for your x.y.zz.com, can you change that to your z.com?

PROFESSOR: No, it says on every dot.

AUDIENCE: OK. Is there a reason that it wasn't made so that you could specify super- or subdomain, but somehow they had to agree on where the information was coming from. So, like, you said some kind of I want to consider all of these to be the same origin as me.

So any of them can attack me. And then you made this symmetric in order for me to impact them as well? [INAUDIBLE] .com means anything that's .com can impact me. And then you put [INAUDIBLE].

PROFESSOR: Yeah, it's tricky. So there's a couple of different answers to that. So first of all, people were very worried about this attack here. So they wanted to make the domain manipulation language be, at least, somewhat easy to understand. So they don't allow more broke settings.

I'll get to one thing in a second, which kind of allows what you're talking about but only with respect to domain [INAUDIBLE]. I'll get to that in one second. And another to mention, too, is that the post message interface does allow arbitrary domains to communicate with each other if they both opt into it. So in practice, people use post message to cross domain communication if they can't set their origins to be the same using these tricks here.

So yeah, so browsers can constrain or widen, I should say, their domain to these suffixes of the original domain. And there's also this little interesting quark here, which is that browsers actually distinguish between a document dot domain value that has been written and one that has not been written, OK.

And there's a subtle reason for this we'll get into in a second. So basically, two frames can access each other if one of two things is true. The first thing is both of the frames set document dot domain to the same value.

And the other way that two frames can access each other is that neither of those frames has changed document domain. And of course, both values have to match. And there's a value match.

So the reason for this is a bit subtle. But the basic idea is that these two rules prevent a domain from being attacked by one of its own buggy or malicious sub-domains. OK? So imagine that you have the domain x.y.z.com. And then, imagine that it's trying to attack y.z.com.

So this guy up here is buggy or evil. So what this guy could try to do is actually shorten his domain to be y.z.com. And then, start messing around with JavaScript state, or cookies or stuff like that here. Right?

So basically, what these two rules over here will say is that if y.z.com does not want to actually allow anyone to interact with it, it will never change it's document.domain value so that when this frame up here does shorten it, the browser will say aha. You've shortened it. You have not. There's a match here in terms of the values.

But this person hasn't indicated they want to opt into this type of chicanery. So does that

makes sense? OK, so that is, basically, how frames work with respect to the same origin policy.

So then we can look at how our DOM node's treated. So DOM nodes, it's pretty straightforward for DOM nodes. So DOM nodes, basically, get the origin of their surrounding frame. Makes sense. Then we can look at cookies. Cookies are complicated and a bit tricky. So cookies have a domain. And they have a path.

So for example, you can imagine a cookie might be associated with the following information. So asterisks dot MIT.edu. And then, 6.858. So you've got this domain thing sitting here, and then, you've got this path thing sitting over here.

So note that this domain can be, possibly, complete suffix of the pages current domain. So you can play, somewhat, similar tricks as we had over there. And note that this path here can actually just be set just to the slash with nothing else there, which indicates that all paths in the domain should be able to have access to this cookie here.

But in this case, we actually have one of these nonempty paths. So whoever sets this cookie, basically, gets to choose what the domain in the path look like. And it can actually be set by the server or can be set on the client side. So on the client side, you can basically right to this JavaScript object called document.cooking.

And there's, sort of, this Byzantine format that you can use to indicate all these paths and things like that. But suffice to say it can be done. So JavaScript can set cookies like this. And also, the server can actually set cookies on HP responses when they come back over the wire. So you can, basically, just use the set cookie header, if you're the server, to set some of these things.

And know that there's also a secure flag that you can set in the cookie to indicate that it's an HTTPS cookie, meaning that HTTP content should not be able to access that cookie. So that's the basic idea behind cookies.

Now note that whenever the browser generates a request to a particular web server, it's going to include all of the matching cookies in that request. So there's a little bit of, sort of, string matching and algorithms that have to take place to figure out what are all the exact cookies that should be sent to the service for a particular request because you can have all these weird, sort of, suffix domain things going on and so on and so forth. But that's the basic idea

behind cookies. So does that all make sense?

AUDIENCE: So can frames access each other cookies if they match those rules?

PROFESSOR: Yeah, so frames can do that. But it's dependent on how the document.domain has been set. And then, it's dependent on what the cookie domain and path have been set.

So yeah, after a bunch of these strained comparisons, yes, frames can access each others cookies if all those tests pass. OK, so yes, that leads me into the next question. So we're trying to figure out how different frames can access each others cookies. So what's the problem? What would be the problem is we allowed arbitrary frames to write arbitrary people's cookies? So what do you think?

Well, it will be bad, suffice it to say. The reason it would be bad is because, once again, these cookies allow the client side of the application to store a per user data. So you can imagine that if an attacker could control or override a users cookie, the attacker could actually, for example, change that cookie for a Gmail to make the user log into the attackers Gmail account.

So when the user logged into the attacker Gmail account, any email that the user typed in could be read by the attacker, for example. You could also imagine that someone could tamper with the Amazon.com cookie. You know, put all kinds of embarrassing ridiculous stuff in your shopping cart, perhaps, or so and so forth.

So cookies are, actually, a very important resource to protect. And a lot of web security attacks try to steal that cookie to do various kinds of evil. So here's another interesting question with respect to cookies. So let's say that you've got the site that's coming from foo.co.uk.

So should the site from this host name be allowed to set a cookie for co.uk? So this is a bit subtle because, according to the rules that we've discussed before, a site from here should be able to shorten its domain, set a cookie for this, and that all seems to be legal. Now of course, as a human, we think this is kind of suspicious because, as a human, we actually understand that this is morally speaking a single atomic domain.

Morally speaking, this is equivalent to .com. The British got screwed. They have to have a dot in there. But that's not their fault. History's unfair. Right? So morally speaking, this is a single domain. So you actually have to have some special infrastructure to get the cookie setting

rules to work out correctly.

So essentially, Mozilla, they have this website called publicsuffix.org. And basically, what this website contains are lists of these rules for how cookies, and origins, and domains should be shrunk given that some things might have dots in them. But actually, they should be treated as a single, sort of, atomic thing.

So actually, when your browser is figuring out how it should do all these various cookie manipulations, it's actually going to consult this side. Or it's going to have this baked in somehow or something like that to make sure that foo.co.uk can't actually just shorten its domain to co.uk. And then, perform some chicanery.

So once again, this is very subtle. And a lot of the interesting web security issues that we find come about because a lot of the original infrastructure was designed just for the English language. You know, for ASCII text or something like this. It wasn't designed for an international community.

So as the internet became more popular, people said, hey, we made some pretty big design decisions here at the beginning. We should actually make this usable on people who use our narrow understanding of what language means. You run into all these crazy problems. And I'll give you another example one of those a later lecture. So does this all makes sense?

OK. So with respect to XML HTTP responses, how are they treated by the same origin policy? So by default, JavaScript can only generate one of these if it's going to its origin server. However, there's this new interface called cross origin request or CORS.

All right, so this is the same origin unless the server has enabled this CORS thing. So basically, this adds a new HTTP response header called access control allow origin. So let's say that JavaScript from foo.com wants to make an XML HTTP request to bar.com.

So that's cross origin, as we described in the rules so far. So if the server in bar.com wants to allow this, it will return in it's HTTP response this header here that's going to say, yes, I allow, for example, foo.com to send me these cross origin XML HTTP request.

The server on bar.com could actually say no. It could refuse the request. In which case, the browser would fail the XML HTTP request. So this is, sort of, a new thing that's come up in large part because of these mash up applications. This need for, somehow, applications from different developers and different domains to be able to share data in some type of

constrained way.

So this could also be asterisks over here if anybody can fetch the data cross-origin, so on and so forth. So I think that's pretty straightforward. So I mean, there's a bunch of other resources we could look at. For example, images.

So a frame can load images from any origin that it desires. But it can't actually inspect the bits in that image because, somehow, the same origin policy says that having different origin directly inspect each others content is a bad thing. So the frame can't inspect the bits.

But it can, actually, infer things like what the size of the image is because it can actually see where the other dominoes in that page have been placed, for example. So this is another one of these weird instances where the same origin policy is ostensibly trying to prevent all information leakage. But it can't actually prevent all of it because embedding inherently reveals some types of information.

CSS has a similar story to images. So a frame can embed CSS from any origin. However, it cannot directly inspect the text inside that CSS file, if it's from a different origin. But it can actually imply what this CSS does because it just can create a bunch of nodes. And then, see how they're styling gets changed. So it's a bit wacky.

JavaScript is actually my favorite example of how this same origin policy struggles to maintain any type of intellectual consistency. So the idea here is that, if you do a cross origin fetch of JavaScript, that is allowed. You can allow that external JavaScript to execute in the context of your own page. You cannot, however, look at the source code for it.

So if you have a script tag source equals something outside your domain, then when that source gets executed, you can call functions in it. But you can't actually look at the JavaScript source code in it. OK, fine. So that seems very nice. However, there are a bunch of holes in this.

So for example, JavaScript is dynamic scripting language. And functions are first class objects. So for any function f, you can just call f.tostring. And that will give you the source code for the function. And people do this all the time. Do things like dynamic rewriting and stuff like that.

So you know the same origin policy doesn't allow you to directly look at the contents of the script tag itself? You can just call this for any public function that that external script has given

you. And just get the source code like that.

Another thing you could imagine doing is you could just get your home server from your domain to just fetch the source code for you. And then, just send it back to you. So oops. I mean, you essentially just asked your home server to run Wget.

And you get the source code that way. OK, so that's, kind of think, goofy. So long story short, the same origin policies here are a bit odd.

AUDIENCE: Presume that par of the reason they do it is to prevent the user from fetching JavaScript because then cookies will be sent as well. So you can get JavaScript tailored to you.

PROFESSOR: Yeah.

AUDIENCE: So if you get your server to fetch it for you, it won't have the user's cookies [INAUDIBLE].

PROFESSOR: That is true. Although, in practice, a lot of times, the raw source code, itself, is not user tailored in practice. But you're right that it will prevent some cookie-mediated attacks like that. Modulo, some of the cookie [INAUDIBLE]. But that's exactly correct.

So because it's actually pretty easy for users and applications to get JavaScript source code, a lot of times, JavaScript source code, when it's deployed, it's actually obfuscated and minified. So if you've ever tried to look and see how a web page works, if you look at the source, sometimes people will do things like move all the white space.

They will also change all the variable names to be super short and have all these exclamation marks. Looks like cartoon characters cursing in the cartoons. So that's, sort of, like a cheat form of digital rights management. But it's all, ultimately, a bit of a crap shoot because you can do things like execute that code in your own browser.

See what it does. Sniff the network. See who it talks to, so on and so forth. But that's, basically, the same origin story for JavaScript. Plug-ins--

AUDIENCE: I was under the impression that the reason you do that is [INAUDIBLE] take less time to download rather than [INAUDIBLE].

PROFESSOR: So that is also a reason they do that, too. That's a good point. But I mean, if you type into the internet, sort of, web page obfuscation or stuff like that, people often try to, somehow, make some type of secrets into either their HTML or their JavaScript.

Maybe they want to obscure the protocol. For example, if the client uses it to talk to the server. Some people will also do the obfuscation for that reason. Pure minification-- in other words, just making the variable names small and moving the [INAUDIBLE] space-- yeah, that's mainly just to save download band, download time.

OK, so that's the story for JavaScript. There's also plug-ins. So this is stuff like Java and things like this. So a frame can easily run a plug-in from either origin. Now plug-ins, depending on who you believe, are actually going to the way of the dinosaurs. Because a lot of the new HTML 5 features, like video tag and things like this, can actually do stuff that you used to only be able to do with a plug-in like Java.

So it's not clear how much longer these things are going to be around. OK, so any questions. OK, so remember that when a browser generates an HTTP request it automatically includes the relevant cookies in that request. So what happens if a malicious site generates a URL that looks like this?

So for example, it creates a new child frame. It says that URL to bank.com. And then, it actually tries to mimic what the browser would do if there was going to be a transfer of money between the user and someone else. So in this URL, in this frame that the attack is trying to create, it tries to invoke this transfer command here.

Say $500. And that should go to the attacker's account at the bank. Now the attacker page, which the user visited because, somehow, the attacker is [INAUDIBLE] go there. What's interesting about this is that, even though the attacker page won't be able to see the contents of this child frame because it's probably going to be in a different origin. The bank.com page will still do what the attacker wants because the browser's going to transfer all the users cookies with this request. It's going to look at this command here and say, oh, the user must've, somehow, asked me to transfer $500 to this mysteriously named individual named attacker.

OK, I'll do. All right, seems reasonable. So that's a problem. Then the reason this attack works is because, essentially, the attacker can figure out deterministically what this command should look like. There's no randomness in this command here.

So essentially, what the attacker can do is try this on his or her own bank account, figure out this protocol, and then just, somehow, force the user browser to execute this on the attackers

behalf. So this is what's called a cross site request forgery. So sometimes you hear this is called CSRF. C-S-R-F.

So the solution to fixing this attack here is that you actually just need to include some randomness in this URL that's generated. A type of randomness that the attacker can't guess statically. So for example, you can imagine that inside the bank's web page it's going to have some form.

The form is the thing, which actually generates request like this. So maybe the action of that form is transfer.cgi. And then, inside this form, you're going to have an input. Inputs are usually used to get in user input like text, key presses, mouse clicks, and stuff like that. But we can actually give this input a type of hidden, which means that it's not shown to the user.

And then, we can give it this attribute. We'll call it CSRF. And then, we'll give it some random value. You know, a72f. Whatever. So remember, this is generated on the server side. So when the user goes to this page, on the server side, it sometimes generates this random here and embeds that in the HTML that the user receives.

So when the user submits this form, then this URL that we have up here will actually have this extra thing up here, which is this token here. So what this does is that this now means that the attacker would have to be able to guess the particular range of token that the server generated for the user each time the user had gone to the page.

So if you sufficient randomness here, the attacker can't just forge one of these things because if the attacker guesses the wrong token, then the server orders will reject your request.

AUDIENCE: Well why should these always be included in the URL and not in the body of the [INAUDIBLE]?

PROFESSOR: Yeah, yea. So HTTPS helps a lot of these things. And there's actually no intrinsic reason why you couldn't put some of this stuff in the body of the request. There's some legacy reasons why forms, sort of, work like this.

But you're correct. And in practice, you can put that information somewhere else in the HTTPS request. But note that just moving that information, for example, to the body of the request, there's still a challenge there, potentially because if there's something there that the attacker can guess. Then the attacker may still be able to, somehow, conjure up that URL. For example, when I'm making XML HTTP request and then, explicitly, setting the body to this thing that the attacker knows how to guess.

AUDIENCE: Well if the attacker just gives you a URL, then that just gets encoded in the header of [INAUDIBLE].

PROFESSOR: If the attacker just gives you a URL. So if you're just setting a frame to URL, then, that's all that the attacker can control. But if you're using an XML HTTP request if, if somehow the attacker can generate one of those, then XML HTTP interface actually allows you to set the body.

AUDIENCE: The XML HTTP request would be limited by, say, an origin. But the attacker could just write a form and submit it. There's nothing [INAUDIBLE] submitting a form like using [INAUDIBLE]. And then, it's sent in the body. But it's still--

PROFESSOR: That's right. So XML HTTP request is limited to the same origin. However, if for example, the attacker can, maybe, do something like this, for example. And the attacker can inject the XML HTTP request here, which would then execute with the authority of the embedded page.

AUDIENCE: Can the attacker [INAUDIBLE] by inspecting the HTML source code?

PROFESSOR: Yes, that's actually a good question. right so it depends on what the attacker has access to. If the attacker-- for example, by doing something goofy like that-- can actually access this JavaScript property called inner HTML. This is a property [INAUDIBLE], right. So if I document that body dot inner HTML, I will get all of the HTML that's inside that page right now.

So yeah. So if the attacker can do this, then yeah. Then you're in trouble. That's right. So a lot of these details, though, depend on exactly what the attacker can and can't do. So it, kind of, makes sense. So if the attacker can or cannot generate Ajax request, that means one thing. The attacker can or cannot look at the right HTML, then you have another thing. So on and so forth.

All right. So yeah. So this is token based thing is a popular way to get around these CSRF attacks. All right, so another thing we can look at are network addresses. So this gets into some of the conversation we've been having about who the attacker cannot contact via XML HTTP request, for example.

So with respect to network addresses, a frame can send HTTP and HTTPS requests to a host plus a port that matches it's origin. But note that the security of the same origin policy is, actually, very tightly tied with the security of the DNS infrastructure because all the same origin policies' rules are based upon what names me.

So if you can control what names me, you can actually want some pretty vicious attacks. So an example of this is the DNS rebinding attack. So in this attack, the goal of the attacker is run attacker controlled JavaScript with the authority of some victim website. We'll just call them victim.com.

So the attacker wants to bus the same origin policies and somehow run code that he has written with the authority of some other site. So here's the approach. So the first thing that the attacker is going to do is register a domain name. So let's say we just call that attacker.com.

Very simple to do. Just pay a couple of bucks. You're ready to go. You own your own domain name. So note that the attacker is also going to set up a DNS server to respond to name resolution requests for objects that reside in attacker.com.

So the second thing that has to happen is that the user has to visit attacker.com. In particular, the user has to visit some website that hangs off of this domain name. This part is actually not tricky.

See if you can create an ad campaign. Free iPad. Everybody wants a free iPad, even though I don't know anyone who's ever won a free iPad. The click on this. They're there. It's in the phishing email, so and so forth. This part's not hard.

So what's going to happen? So this is actually going to cause the browser to generate a DNS request to attacker.com because this page has some objects that refer to some objects that live in attacker.com. The browser's going to say I never seen this domain before. Let me send the DNS resolution request to attacker.com.

So what's going to end up happening is that the attackers DNS server is going to respond to that request. But it's going to respond with a DNS result that has a very short time to live. OK? Meaning that the browser will think that it's only valid for a very short period of time before it has to go out and revalidate that. OK?

So in other words, the attacker response has a small DTL. OK, fine. So the user gets the response back. The malicious website is now running on the user side. Meanwhile, while the user's interacting with the sight, the attacker is going to configure the DNS server that he controls.

The attacker is going to bind the attacker.com name to victim.com's IP address. Right? So

what that means is that now if the user's browser ask for a domain name resolution for something that resides in attacker.com, it's actually going to get some internal address to victim.com.

This is actually very subtle. Now why can the attacker's DNS resolver do that? Because the attacker configures it to do so. The attacker's DNS server does not have to consult victim.com to do this rebinding.

So perhaps, you can see some of the outline in the attack now. So what will happen is that the website wants to fetch a new object via, let's say, AJAX. And it thinks that that AJAX request is going to go to attacker.com somewhere externally. But this AJAX request actually goes to victim.com.

And the reason why that's bad is because now we've got this code on appliance side that resides on the attacker.com web page that's actually accessing now data that is from a different origin from victim.com. So once this step of the attack completes, then the attacker.com web page can send that contact back to the server using [INAUDIBLE] or do other things like that. So does this attack make sense?

AUDIENCE: Wouldn't it be more sensible to do the attack the other way around? So to [INAUDIBLE] victim.com to the attackers IP address. Because that way you're the same origin as victim.com so you can get all the cookies and such.

PROFESSOR: Yeah, so that would work, too, as well. So what's nice about this though is that, presumably, this allows you o do nice things like port scanning and stuff like that. I mean, your approach will work, right. But I think here the reason why you do--

AUDIENCE: [INAUDIBLE].

PROFESSOR: Because, essentially, you can do things like constantly rebind what attacker.com points to to different machine names and different ports inside of victim.com's network. So then, you can, sort of, step through. So in other words, let's say that the attacker.com web page always thinks it's going to attacker.com and issuing an AJAX request there.

So every time the DNS server rebinds, it [INAUDIBLE] to some different IP address inside of victim.com's network. So it can just, sort of, step through the IP addresses one by one and see if anybody's responding to those requests.

AUDIENCE: But the client, the user you're attacking, doesn't necessarily have inside access to victim.com's network.

PROFESSOR: So what this attack, typically, ensues is that there are certain firewall rules that would prevent attacker.com from outside the network from actually looking through each one of the IP addresses inside of victim.com. However, if you're inside corp.net-- if you're inside the corporate firewall, let's say-- then machines often do have the ability to contact [INAUDIBLE].

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, yeah. Exactly.

AUDIENCE: Does this work over HTTPS?

PROFESSOR: Ah, so that's an interesting question. So HTTPS has these keys. So the way you'd have to get this to work with HTTPS is if somehow, for example, if attacker.com could-- let me think about this.

Yeah, it's interesting because, presumably, if you were using HTTPS, then when you sent out this Ajax request, the victim machine wouldn't have the attackers HTTPS keys. So the cryptography would fail somehow. So I think HTTPS would stop that.

AUDIENCE: Or if the the victim only has things on HTTPS?

PROFESSOR: Yeah. So I think that would stop it.

AUDIENCE: If you configure the [INAUDIBLE] use the initial or receiving result [INAUDIBLE]?

PROFESSOR: That's a good question. I'm actually not sure about that. So actually, a lot of these attacks were dependant on the devil in the details, right? So I'm not actually sure how that wold work.

AUDIENCE: It uses the first domain.

PROFESSOR: It would use the first domain? OK. Yep?

AUDIENCE: So why can the attacker respond with the victims IP address in the first place?

PROFESSOR: So why can't-- what do you mean?

AUDIENCE: [INAUDIBLE]. Why has the attacker team [INAUDIBLE] has to respond with the attacker's IP

[INAUDIBLE]?

PROFESSOR: Oh, well, yeah. Since the attacker has to, somehow, get it's own code on the victim machine first before it can then start doing this nonsense where it's looking inside the network. So it's that initial step where it has to put that code on the victims machine.

All right, so in the interest of time, let's keep moving forward. But come see me after class if you want to follow up the question. So that's the DNS rebinding attack. So how can you fix this? So one way you could fix it is so that you modify your client-side DNS resolver so that external host names can never resolve to internal IP address.

It's, kind of, goofy that someone outside of your network should be able to create a DNS binding for something inside of your network. That's the most straightforward solution. You could also imagine that the browser could do something called DNS pinning. Whereby, if it receives a DNS resolution record, then it will always treat that record as valid for, let's say, 30 minutes, regardless of whether it has a short TTL set inside it because that also prevents the attack, as well.

That solution is a little bit tricky because there are some sites that actually, intentionally, use dynamic DNS and do things like load balancing and stuff like that. So the first solution is probably the better one. OK, so here is, sort of, a fun attack. So we've talked about a lot of resources that the origin protects-- the the same origin policy protects. So what about pixels?

So how does the same origin policy protect pixels? Well as it turns out, pixels don't really have an origin. So each frame gets its own little bounding box. Just a square, basically. So a frame can draw wherever it wants on that square.

So this is, actually, a problem because what this means is that a parent frame can draw atop of it's child frame. So this can lead to some very insidious attacks. So let's say that the attacker creates some page.

And let's say, inside of that page, the attacker says click to win the iPad. The very same standard thing. So this is the parent frame. Now what the parent frame can do is actually create a child frame that is actually the Facebook Like button frame.

So Facebook allows you to run this little piece of Facebook code you can put on your page. You know, if the user clicks Like, then that means that it'll go on Facebook and say, hey, the user likes the particular page. So we've got this child frame over here. That actually turned out

remarkably well.

Anyway, so you've got this Like thing over here. Now what the attacker can do is actually overlay this frame on top of the click to get the free iPad and also make this invisible. So CSS let's you do that. So what's going to happen? As we've already established, everybody wants a free iPad.

So the user's going to go to this site, click on thing-- this area of the screen-- thinking that they're going to click here and get the free iPad. But in reality, they're clicking the Like button that they can't see that's invisible. It's like layered atop the C index.

So what that means is that now maybe they go check their Facebook profile, and they've liked attacker.com. You know, and they don't remember how that happened. So this is actually called click jacking attack because you can imagine you can do all kinds of evil things here. So you can imagine you could steal passwords this way.

You could get raw input. I mean, it's madness. So once again, this happens because the parent, essentially, gets the right to draw over anything that's inside this bounding box. So does that attack make sense? Yeah.

AUDIENCE: [INAUDIBLE], what do you mean the parent gets to draw over anything [INAUDIBLE]?

PROFESSOR: So what I'm trying to indicate here is that, visually speaking, what the user just sees is this.

AUDIENCE: Oh, that's the parent frames.

PROFESSOR: Yeah, this is the parent frame. That's right. This is the child frame. So visually speaking, the user just sees this. But using the miracle of my da Vinci style drawing techniques, this is actually overlaid atop this transparently. So that's the child frame. That's the parent frame.

OK so, there's a couple different solutions-- you can imagine-- for solving this. The first solution is to use a frame busting code. So you can actually use JavaScript expressions to figure out if you have been put into a frame by someone else.

So like, one of these tests is you compare the reference self to top. So in the JavaScript world, self refers to frame that you yourself aren't in. Top refers to the frame at the top of the frame hierarchy. So if you do this test and you find out that self is not equal to top, then you realize that you are a child frame. And then you can refuse to load or do things like this.

So this, in fact, is what will happen if you try to create a frame for, let's say, CNN.com. You can actually look in the JavaScript source and see that it does this test because CNN.com doesn't want other people taking credit for it's content. So it only wants to be the top most frame. So that's one solution you can use here.

The other solution that you can use here is also to have your web server send this HTTP response hitter called x-frame options. So when the web server returns a response, it can set this header. And it can basically say, hey, browser, do not allow anyone to put my content inside of a frame. So that allows the browser to do the enforcement.

So that's pretty straightforward. So there's a bunch of other, sort of, crazy attacks that you can launch. Here's another one that's actually pretty funny. So as I was mentioning before, the fact that we're now living in a web that's internationalized actually mean that there's all these issues that come up involving name and how you represent host names.

So for example, let's say that you see this letter right here. So what does this look like? This looks like a C, right? What is this? A C in ASCII in the Latin alphabet? Or is this a C in Cyrillic? Hard to say, right? So you can end up having these really strange attacks where attackers will register a domain name, like cats.com, for example. But this is a Cyrillic C.

So users will go to this domain. They might click on it or whatever thinking they're going to Latin alphabet C, cats.com. But instead, they're going to an attacker one. And then, all kinds of madness can happen from there, as well.

So you might have heard of attacks like this are like typo squatting attacks where people register for names like F-C-E book.com. This is a common fumble finger typing for Facebook.com. So if you control this, you're going to get a ton of traffic from people who think they're going to Facebook.com.

So there's a bunch of different, sort of, wacky attacks that you can launch through the domain registry system that are tricky to defend from first principles because how are you going to prevent users from mistyping things, for example? Or how would the browser indicate to the user, hey, this is Cyrillic? Is the browser going to alert the user every time Cyrillic fonts are included?

That's going to make people angry if they actually use Cyrillic as their native font. So it's not quite clear, technologically speaking, how we deal with some of those issues. So yeah, there's

a bunch of other security issues that are very subtle here.

One thing that's interesting is if you look at plugins. So how do plugins treat the same origin policy? Well plugins often have very subtle incompatibilities with the rest of the browser with respect to the same origin. So for example, if you look at a Java plug-in, Java, oftentimes, assumes that different host names that have the same IP address actually have the same origin.

That's actually a pretty big deviation from the standard interpretation of the same origin policy because this means that if you have something like x.y.com and, lets say, z.y.com, if they map onto the same IP address, then Java will consider these to be in the same origin, which is a problem if, for example, this site gets [? owned ?] but this one doesn't.

So there's a bunch of other corner cases involving plug-ins. You can refer to the tangled web to see some more about some of those types of things. So the final thing that I want to discuss-- you can see the lecture notes for more examples of a crazy Attacks that people can launch-- but the final thing that I want to discuss is this screen sharing attack.

So HTML 5 actually define this NEW API by which a web page can allow all the bits in it's screen to be shared with another browser or shared with the server. This seems like a really cool idea because now I can do collaborative foo. We can collaborate on a document at the same time. And it's exciting because we live in the future.

But what's funny about this is that, when they designed this API, and it's a very new API, they apparently didn't think about same origin policies at all. So what that means is that if you have some page that has multiple frames, then any one of these frames, if they are granted permission to take a screenshot of your monitor, it can take an entire screen shot of the entire thing, regardless of what origin that other content's coming from.

So this is, actually, a pretty devastating flaw in the same origin policy. So there's some pretty obvious fixes you can think about. So for example, if this person's given screenshot capabilities, only let it take a screenshot of this. Right? Not this whole thing.

Why didn't the browser vendors implement it like this? Because there's such pressure to compete on features, and to innovate on features, and to get that next new thing out there. So for example, a lot of the questions that people were asking about this particular lecture online [INAUDIBLE] was like, well, why couldn't you do this? Wouldn't this thing make more sense?

It seems like this current scheme is brain dead. Wouldn't this other one be better? And the answer is, yes. Everything, yes. That's exactly correct. Almost anything would be better than this. I'm ashamed to be associated with this.

But this is what we had. So what ends up happening is if you look at the nuts and bolts of how web browsers get developed, people are a little bit better about security now. But like, with the screen sharing thing, people were so pumped to get this thing out there, they didn't realize that's it's going to leak all the bits on your screen.

So now we're at his point with the web where-- I mean, look at all these things that we've discussed today. So if we were going to start from scratch and come up with a better security policy, what fraction of websites that you have today are going to actually work? Like, approximately, .2% of them. Right?

So users are going to complain. And this is another constant story with security. Once you give users a feature, it's often very difficult to claw that back, even if that feature is insecure. So today, we discussed a lot of different things about the same origin policy and stuff like that. Next lecture, we'll go into some more depth about some of those things we talked about [INAUDIBLE].