Course Overview

Hi, this is Troy Hunt, and welcome to my course on what every developer must know about HTTPS. I've created dozens of plural site courses over the years on all sorts of aspects of web security, so I spend a huge amount of my time doing that and traveling around the world talking to developers in all sorts of different organizations about how to secure their things on the web. Some of the things you'll learn in this course is why we need HTTPS. We run through a bunch of the fundamentals, about all sorts of different positive things that HTTPS does before we dive into what you actually need to do to secure your apps. You'll also learn about what many people perceive as barriers, so HTTPS adoption, and then you'll see why they simply don't apply anymore. I go through a lot of examples in industry precedents that will contextualize just why HTTPS is so important. You'll see where it's all gone wrong before and learn how to implement this essential security defense properly. By the end of this course, you'll be all set to roll out HTTPS in your app, and if you already have it, you'll almost certainly learn a bunch of new tricks to make it faster, more secure, and more reliable. No matter what platform you're working on, this is a great course for learning everything you need to know as a developer about properly implementing HTTPS.

The HTTPS Value Proposition

Overview

Hi, this is Troy Hunt and welcome to What Every Developer Must Know About HTTPS. This is a fairly self-explanatory title. And what this course is going to help you understand is that HTTPS is a lot more nuanced than just having a padlock in the address bar of the browser. In fact, there are many subtleties to the way HTTPS is implemented by developers in web applications. There are many pitfalls that are easy to get wrong. And likewise, there are many additional layers of security that you can employ at the actual application level. So these are things that are entirely in control of those of you who are actually building web software. In this first module, I want to talk about the HTTPS value proposition. Let's go and have a look at the overview. The first thing that we're going to look at in this module is just how much HTTPS is increasing in usage. It's moving forward very rapidly. And HTTPS is becoming an essential component of any software running on the web. It's no longer just the domain of banking or finance. HTTPS is something we need to start getting on every website as soon as possible. As we progress through this module, you'll see how HTTPS does play a pretty essential role in software that we build for the modern web. Now many people think that HTTPS and the padlock in the address bar is just about privacy, so protecting data from prying eyes. But it's actually about a lot more than that. We're going to talk about other really valuable attributes of

HTTPS in this module. And finally, we are moving towards HTTPS by default, which means having HTTPS everywhere. The browser vendors, in particular, are really driving this move forward. So now's the time to start getting ready for HTTPS if you're not using it already on everything. And if you are, then now's a great time to strengthen your implementation at the application layer. So that's what we're going to cover in the first module. Let's move on and start looking at the rise and rise of HTTPS.

The Rise and Rise of HTTPS

This was a great result out of 2016 where for the first time, we saw more than half the page loads that Mozilla telemetry captured occurring over HTTPS. 2016 became the tipping point where more content was loaded securely than that which was loaded insecurely. Now this doesn't mean that more than 50% of websites are using HTTPS, far from it. And we'll look at some of those statistics in a moment. But when you consider the HTTPS usage of really dominant players on the web, players like Gmail, Facebook, Twitter, they've implemented HTTPS across everything. And obviously they have huge audiences that spend a large part of their day on those sites, loading secure pages, just like we see in the quote here. It's not just platforms like banking, or as I just mentioned, email and social media, though, that are using HTTPS. Let me give you another example. Here's the New York Times. And you can see the date on the paper at the time I'm recording this is Wednesday, January 11, 2017. This is a really significant date for the New York Times because it's the day that they went to HTTPS by default. We have a padlock in the address bar. We have the word Secure. We have HTTPS scheme in the URL. This is a newspaper, we're reading stories. You're not necessarily providing sensitive data like passwords. Yet the New York Times has elected to go HTTPS only. And later on, we're going to talk about some of the reasons why every single website that's not already operating securely needs to start thinking about following in their footsteps. The good news is that many are. And the uptake of HTTPS is increasing dramatically. Let me show you some figures that demonstrate just how rapidly we're moving forward with HTTPS. This is a great little graph that shows the uptake of HTTPS across the Alexa Top 1M Websites. This data was prepared by fellow security researcher Scott Helm, and what Scott does is every six months, he scans the top one million Alexa websites, and he looks at various aspects of their security implementation. Now one of those aspects is about whether the site forces the user to load content over HTTPS. So if you try and make an HTTP connection, it redirects you to the secure scheme. The great thing about this graph is that we see across a period of only 12 months, the uptake more than doubles. We've gone from 6.7% of that Alexa Top 1M, forcing HTTPS to 13.6%. Now on the one hand, that's fantastic, we've doubled it in a year. On the other hand, it's only 13.6%. And that means we've still got a long way to go. So HTTPS is increasing dramatically.

Let's go and start having a look at why HTTPS is important. Let's talk about man in the middle attacks.

Understanding a Man in the Middle Attack

Just before we start getting into the mechanics of man in the middle attacks, and indeed before we start talking about the technology behind HTTPS, let's just be clear about who this course is for and per the title here, it is intended to be for developers. So what I'm talking about here is the individuals that are actually building the software that need to use HTTPS, that need to implement those secure communications. So this is predominantly people building web applications. Now we still have Sys Admins, and Sys Admins are very important people. When it comes to HTTPS they're often involved in the procurement of certificates, the configuration of them on web servers, and of course actually keeping them current, renewing them on a regular basis. Now you may well play both these roles. Many of us do, we have to wear multiple hats in our jobs. But I want to be clear that we're going to focus on that role on the left, on the role of the developer. And we're not going to spend much time delving into Sys Admin world and the role that they play in managing certificates. With that said, let's move on and look at that man in the middle attack. Let's go through the mechanics of this class of attack, because this is really the reason why we have HTTPS. Now, a man in the middle attack, as a concept, means that we have Person A and Person A wants to talk to Person B, and they have this communication going backwards and forwards. Now that's fine, the two individuals are having this discussion, until a Man in the Middle comes along. This Man in the Middle is able to intercept the conversation. Very shortly, we'll talk about some of the things the Man in the Middle can do when they do intercept those communications. But typically what'll happen is they'll then forward that communication on to the original destination person, Person B in this case. The response will come back and it will go back through the Man in the Middle, back to Person A. And what we're talking about with this class of attack is that Person A and Person B don't necessarily know that the Man in the Middle is there. All they know is they're having a conversation. But the contents of that conversation are not being protected as they travel across the communication protocol. Now, as we talk about HTTPS, Person A is inevitably going to be someone with a browser or some other rich client. It could well be a mobile app, for example, or an IOT device. They need to talk over HTTPS, as well. Now Person B in this case may actually be a website. It might not be a person at all. It's still a man in the middle attack. We've still got two parties trying to have a conversation and someone jumping in the middle of it. It's just that in this case, that communication is going out over HTTP. And of course, because there is no S, it is an insecure transport layer, hence the man in the middle attack is possible. But how does this happen? How does the man get in the middle in the first place? Let's go and have a look. When we talk particularly about the web and how a man in the middle attack works, there are multiple points where the attacker may insert themselves into the communication. So for example, let's imagine we have a browser at one end. Now a browser is going to be running on say a mobile device. And it's going to talk to a wireless router. It may be a desktop PC plugged into Ethernet, also talking to a router somewhere. The point is that we have the communication channel here between the device and what is the first intermediary in the chain of communication. Now imagine an attacker has control over that wireless router. You're sending all of your traffic through the device that is in their grasp. What if they're saving all of the packets that go backwards and forwards through the router? What if they're using the router to redirect your traffic to somewhere else? If we put this in a real world context, if you go to a cafe and you connect to the free cafe Wi-Fi, how do you know what they're doing with your traffic? Rogue wireless hotspots is a serious risk when it comes to man in the middle attacks. Let's go upstream a little bit. The wireless router has to connect to an ISP so that it can go back out over the internet. There are many, many different cases of ISPs intercepting traffic and performing what is effectively a man in the middle attack. I'm going to show you one of those from an ISP very shortly. So this is another point of trust. And of course everything between the wireless router and the ISP is also subject to potential man in the middle attacks. Now that may be the copper wire going out of your house to an exchange somewhere. It might be the fiber optics which then run from that exchange to the ISP. The point is is that these communications do pass through multiple points and they are subject to potential man in the middle insertion. Let's continue. The ISP at some point needs to route that communication through to the target web server. Now this could mean sending the traffic over huge distances. It could be sending it to the other side of the world, through other countries, through other exchanges, through other points where someone who wants to get into the middle of the communication can do so. Now at that level, we're talking more about the likes of say nation states. A government wants to see all of the traffic that's coming and going through their infrastructure. That's the sort of adversary we need to think about when we consider how ISPs are communicating upstream. So that's a fairly basic look at the different points where you might have a man in the middle. What I want to do now, though, is talk about three different attributes of the communication that a man in the middle might want to compromise. And I'm not just going to talk about them hypothetically. For each one of these, I'm going to give you industry precedents where we've seen these things exploited in the wild. Let's go and start with confidentiality.

The Importance of Confidentiality

Confidentiality is what most people think of when they think about HTTPS. So they're thinking about how do I keep the things in my communications secret? So for example, how do I protect my passwords when I log in? Let's say I'm logging into my bank. I want to make sure that all my account balances and other banking details are protected from someone who might sit in the middle of the communication. Another common one is email. If you fire up your browser and go to Gmail, you want to know that your communication is being encrypted and no one else is reading the contents of those HTTP requests as you browse through your mail. Now these are fairly obvious ones, but let me go and give you another example of a piece of data that's sent over the internet that's enormously important to protect. And I'm going to give you an industry precedent here. In 2010, we saw the emergence of a Firefox plug-in called Firesheep. And what Firesheep allowed you to do is you could go to an internet cafe and observe the unencrypted packets of other people in the same cafe, or any other wireless hotspot, but cafes are sort of the chronicle example. Now within those packets, if someone was logged onto Facebook, back at that time Facebook did not protect their authentication cookies. So you could be sitting there in the cafe, watching the authentication cookies of other people fly around the network. And the Firesheep plug-in allowed you to identify those cookies, steal it, and hijack their session. Now if the concept of session hijacking is foreign to you, go and have a look at my course on ethical hacking session hijacking. There's an entire course there that explains how that works. And it also talks about the value of protecting them using HTTPS. So Firesheep was a really important watershed moment that forced Facebook and many others doing similar things to consider authentication cookies as a piece of sensitive data and always protect them using HTTPS. So that's confidentiality, bits of the communication that we don't want to leak to other people. Let's go and look at integrity.

The Importance of Integrity

Integrity is one that many people don't initially consider when they think about HTTPS, but it's enormously important. Let me give you an example. Imagine you go to a web page that loads a login form over HTTP. It's going to post it to HTTPS, so we'll have confidentiality of the password when the form is submitted. But because the form loads insecurely, it can be modified. And if you can modify it, attackers can do things like change the form action. So imagine now you load the form insecurely. It's meant to submit to HTTPS. But because you have no secure transport layer, you can't trust the integrity, and the attacker changes where it posts to. Malicious tracking is another example. If you have a man in the middle that's able to insert things into the contents of the response, then they now have the ability to start tracking your movements. A little bit later on this module, I'm going to show you some abuses of unencrypted traffic and you'll see an ISP doing something somewhat similar to this. Injecting of malware is another example. If there's a router that's compromised which can modify traffic as it passes through there, there's every possibility for an attacker to add malicious code into the response. Now something like this is important, no matter how sensitive the data on the page is. This is the sort of thing that's driven the New York Times to go HTTPS only, because once it's encrypted, it doesn't matter if the router or the ISP's compromised. They can't modify the contents of the traffic, because now you have integrity. Let me give you an example of where we've seen this exploited in the world. Here's another example from many years ago before Facebook was loading their login pages over HTTPS. Back in 2011, they were loading the login page insecurely. It was posting to HTTPS, so your password was protected. When the form was submitted, you had confidentiality. But you didn't have integrity, because the login page itself loaded insecurely. Now in Tunisia where they were having a lot of political uprising at the time, the government decided that they wanted to get access to the Facebook credentials of their citizens. So they literally injected a keystroke logger into the Facebook login page. Now they could do that because it wasn't loaded over HTTPS. There was no guarantee of integrity when that content was loaded. And this again was another one of those watershed moments where the industry realized that we needed to do better. Facebook certainly no longer does this. Many other sites though, still do. And we'll talk about the increasing problems they face a little bit later on. For now, though, let's go and look at one more attribute of HTTPS, and that's authenticity.

The Importance of Authenticity

Authenticity is enormously important because it gives us confidence that we are actually talking to who we think we are. Now how's that important as it relates to HTTPS? Well, a fundamental part of the communication is that you're presented with a certificate from the server that gives you confidence in the identity of who you're talking to. We'll go into the mechanics of that a bit more in the next module. For now, though, the reasons that's important is because it prohibits attacks such as DNS hijacking. So for example, someone poisons the DNS and causes host names to resolve to different IP addresses and then they serve up their own rogue websites from those addresses. If it's just an HTTP website, they can do that all day long. However, if it's HTTPS, then a valid certificate must be served. And this is where the authenticity comes in. Phishing attacks are another one that HTTPS can really help mitigate, due to the authenticity requirement of certificates. If you receive an email allegedly from PayPal, and it's asking you to reset your account, and you go to the site, that site won't be able to serve up a valid certificate for PayPal. Now there are some very important nuances here. The site could serve up a valid certificate for a totally different domain. But then a site like PayPal has an extended validation certificate with the organization's name in it, which is something that you can't spoof. We'll talk more about that later on in the course. But certainly the authenticity attribute of certificates is a very positive thing when it comes to mitigating the risk of phishing attacks. Malicious host file entries is another one. And these host file entries instruct your device where to resolve host names to. We've seen attacks in the past which have added malicious host name entries such that when you make a request for say your bank, instead of going to your bank, the request is sent off to a totally different website. Even though you see your bank's address in your browser, using HTTPS and requiring a valid certificate puts a stop to that. You can no longer simply route the traffic through to a totally different website, because it won't have the right certificate and it won't satisfy the authenticity requirement. We'll look more at how certificates are exchanged in the next module. For now, though, let me give you one more industry precedent that demonstrates the value of authenticity. I could've shown you any one of dozens of different stories about how home routers are being compromised and then DNS modified to route traffic through to malicious sites. This is an enormously prevalent attack. And we often see it exploiting vulnerabilities, such as cross-site request forgery risks in the router's administration software. I talk about CSRF risks in detail in my Hack Yourself First course, How to Go on the Cyber Offense. So if that's an unfamiliar term, go and check out the Cross-Site Attacks Module in that course. For now, though, that rounds out the three attributes that are really important about HTTPS. Confidentiality, integrity, and authenticity. So we've now defined those and looked at an industry precedent for each one. Abuses of unencrypted traffic are everywhere. We've just looked at three examples there. Let me go and show you some more, though, just to be certain that we are absolutely confident that we really do need HTTPS everywhere.

Abuses of Unencrypted Traffic Are Everywhere

I want to show you several other abuses of unencrypted traffic just to reinforce how important it is for us to get HTTPS everywhere, and I want to begin with this tweet. Someone tweeted this one to me before I rebuilt my blog and forced everything over HTTPS. So this was back in the day when it loaded over HTTP. Now, like many sites, I have gone through this process of having something which was legacy served over HTTP and then come to the point where it really made a lot more sense to serve it securely. In this case, David is on a Norwegian Airline's flight, and he's using the free in-flight Wi-Fi. And what Norwegian has done, and David's drawn some little arrows pointing to this, is they've added that red banner to the top of the page. And you can see there's a little airplane there, and the airplane moves across the screen as you're flying through the air. It tells you how far you've got to go. They are reading the contents of your communication, and they're modifying it literally on the fly. Now, ironically, David was reading a blog post over it about VPNs, and if he had of been using a VPN, then Norwegian wouldn't have been able to do this. Also, if I had of served the page over HTTPS, Norwegian wouldn't have been able to do this. We would have had integrity. Let me give you one more example. How many times have you connected to free Wi-Fi, particularly at an airport or a hotel, and you've opened your web browser, and a whole bunch of your tabs have suddenly ended up on the captive portal of the network you've connected to? You've got dozens of tabs, and they're all saying please agree to the terms and conditions before you can continue to use the Wi-Fi. Now this is effectively DNS spoofing. They're taking a request which your browser has made to an HTTP address, and instead of returning the content for that website, they've returned their own content. They can only do that because it's an insecure request. And the next time you do this, have a look at your tabs, and you'll see that every HTTPS tab will have a connection error, whereas every HTTP tab gets spoofed, and you see the correct URL in the address bar, yet the content of the Wi-Fi network. That again is a serious issue because they're actually taking that request, rerouting it, and directing it to a totally different website. As we progress through this course, the significance of all of these incidents will become clearer. For now though, let's just another few examples of how unencrypted traffic is abused. Let's move on now, and I want to talk about some of the reasons why people believe it may be difficult to move their website to HTTPS.

The (Perceived) Barriers to HTTPS

Let's talk about some of the barriers that people perceive when it comes to moving their website to HTTPS. Now, we are going to talk about how to overcome every single one of these, as we go through the course. But it's only fair to acknowledge the concerns that many people do have. So for example, concerns such as cost. People are concerned not only that they need to go and buy a certificate, and I'm going to give you some examples of how to get them for free a little bit later on, but they're also worried about the potential overhead on the infrastructure. Now that's just simply not an issue in modern-day implementations of HTTPS. It's a little bit more of that Sys Admin space, so we're not going to drill into it in this course, but we will go and tackle the perception that you need to pay money for certificates. Another perceived barrier is complexity. People say, "Well, it's hard to set up." I just mentioned Sys Admins again, and they would normally take responsibility for the setup of certificates. But I also said earlier on that many times we have to wear both hats. You are the technology person. And you've got to both build the application and make the certificates work. Now certainly there can be complexities, but equally there are services available today that can make this very, very easy, all the way to the point where you don't need to know anything about configuring HTTPS on your actual web server. We're going to look at those later on in the module specifically about overcoming these barriers. Speed is another one. There's a long-held belief that if you start encrypting and decrypting data, you're going to pay a performance penalty. And later on in the course, I'm going to show you how it can actually be quite the reverse. There are attributes of HTTPS that can make your application go faster than what it ever would with only HTTP. And finally, compatibility. And this is a concern which was valid at a point in time, but in the modern day is not such an issue. And what people were really concerned about here is that if they start serving their content over HTTPS, they need every other provider who inserts things into their page to also serve them over HTTPS. So for example, ad networks. If you run a website with ads on it, and you want your website to be HTTPS, then everything that you insert into that page must also be HTTPS, therefore then maybe a compatibility barrier. A little bit later on, when we get into securing the application, we'll talk about what happens when that embedded content isn't served securely. So what happens when compatibility goes wrong? The good news today, though, is just about every provider of services that need to be embedded in other people's applications do support HTTPS, simply because it is growing in prominence so much that they need to. Customers are demanding it. So there are four of the most common barriers we see to adopting HTTPS. Let's go and look at one more thing, and it's the big browser shift.

The Big Browser Shift

This is the Qantas website. It's a site that I spend quite a lot of time on. And one of the things that always frustrates me somewhat is that as you can see here, it's asking for my login details, my membership number, my last name, my pin. And it served this up over an HTTP connection. Now this is really the problem we saw before with Tunisia and Facebook. It's an integrity problem. When this page loads, I don't have any confidence whatsoever that it hasn't been modified by a man in the middle somewhere between me and the web server. And that's a particularly bad problem for a website used by travelers who are often in locations with free Wi-Fi where you're going to be more worried about a man in the middle. Now this is Chrome 55, but then a change came along, and Chrome 56 started doing this. Now this may be quite subtle, so let me highlight what's different. And it's this section just here in the address bar. As of Chrome 56, it now explicitly tells you that the web page with a login form loaded over HTTP is not secure. Let's have a look at what happens when we load a site over HTTPS. So for example, my own blog. Now we have a very obvious visual indicator here that the site has loaded securely. We've got a green padlock, we've got the word secure, and we've explicitly got the scheme showing HTTPS. But compare that to the Qantas login page on the left of the screen in Chrome 55. It doesn't tell you that it's insecure. All you have in this case is a little i icon next to the URL. So have a look at these two together. We're seeing the secure page explicitly tell you that it's secure, yet the insecure page is not telling you anything at all. Now when we look at Chrome 56 to the right of the screen, it's a lot more explicit. It's telling you that this page is not secure. Now that's going to make individuals think much more carefully before entering their credentials. And it's also going to make websites like Qantas really start to question why they don't have HTTPS on their login page. It's not just Chrome, either. We're seeing the same thing coming to Firefox. And this is a good thing. It's a slightly different implementation. We're actually seeing it here right next to the username field rather than in the address bar. But it's the same premise. The browsers are holding the websites more accountable to serve content securely. And this is a really positive thing. But this is only one step along the way in this big browser shift, because here's what they're going to do next. So imagine that Not Secure text that we saw just before on the Qantas login page in Chrome 56. Imagine that is on every single HTTP website. What that means is individuals are going to start being told that every single website that is not HTTPS is insecure. Imagine what that's going to mean for a website like this. This is the HSBC Bank at the time of recording. And many banks do this in early 2017. They load the homepage over an insecure connection. Now at the moment, Chrome 56 is just putting the little info icon there in the URL. But imagine what it would do to consumer confidence if you went to your bank and the browser explicitly told you that it is not secure. Now we haven't been given a timeline for this by the likes of Chrome and Mozilla. And it's not something that they can do too early, either. They want to get more adoption of HTTPS first, because if people see a security warning too many times it's just going to desensitize them. But they are being very clear right now. And in fact, Chrome has said quite explicitly that you should be planning to migrate everything to HTTPS for every single page. And they're going to hold sites that don't do that accountable by showing that Not Secure message. Browsers are playing a really active role in forcing websites to move forward and implement HTTPS. And that is a really good thing. So that's it for the HTTPS Value Proposition. Let's go and summarize the module.

Summary

Let's summarize the module, and one of the first things we looked at was the different points at which a man in the middle might insert themselves into the communication. So for example, when the device is talking to a Wi-Fi router or it may happen further upstream a bit as the router talks to the ISP. Or possibly even on the internet backbone, particularly via the likes of nation states whilst your ISP is routing traffic to the destination server. I took you through those potential points to try and establish the fact that there are multiple ways that these attacks can happen. We then went on and had a look at these three attributes of HTTPS. Confidentiality, so keeping the contents of the communication secret. Integrity, making sure it's not modified. And authenticity. So having confidence that the client is talking to the server that they think they're talking to. And we looked at abuses of all these three things, as well. So Firesheep being used to steal authentication cookies for Facebook, the Tunisian government modifying Facebook login pages and routers being compromised and having their DNS changed in order to send traffic to malicious websites. So these attacks do happen. This is not a hypothetical risk. This is real world stuff. We also looked at different places where unencrypted traffic is being exploited. Comcast, the ISP injecting messages about how much of your data has been used. Norwegian Airlines putting a little airplane on the screen for your convenience, yet still actually reading and modifying your traffic. And of course those hotel and airport Wi-Fi hotspots hijacking every single browser tab to give you a login page. All of those things are frustrating experiences. But the good news is, as we looked at earlier in the module, HTTPS is getting a lot more traction. Mozilla reporting that more than half of the websites loaded were HTTPS. Scouts reports showing that the adoption of HTTPS had doubled across the Alexa Top 1M websites in only one year. These are really good signs, and I'm hoping that this course helps you increase those numbers. So let's move on. Let's start moving towards getting your site secure. Let's go and have a look at the next module on HTTPS fundamentals.

HTTPS Fundamentals

Overview

In this module, I want to start talking about some of the HTTPS fundamentals. So these are some of the things that you have to know in order for the coming modules to really make sense. We're not going to go too deep and, again, this is intended to be everything that developers must know about HTTPS; but I do also want to touch briefly on the mechanics of how HTTPS communications work. Let's go and have a look at the overview of the module. This module is going to be all about understanding some of the really fundamental things you need to know about HTTPS when you're developing web applications. So, for example, it's very important to understand what a certificate authority, or CA, is. These are the entities responsible for issuing the certificates which are then used for HTTPS communication. Now, when we talk about HTTPS, we can't really have that discussion without talking about SSL and TLS. These are similar, but different; and the terms are regularly used interchangeably, and there are some reasons behind that. So we're going to take a look at what is the difference between SSL and TLS, how do we normally refer to the two, and then what should we actually be saying when we talk about secure communication over the internet? I also want to touch on developing and debugging with HTTPS. It's one thing to get a certificate from a certificate authority on a live running website; yet it's another thing to get one for your own development purposes on your local machine. I'm going to show you how to generate one of those certificates very quickly and very easily. And, finally, I want to show you a fantastic site for testing bad HTTPS implementations; and this will start to give you a bit of an idea of just how nuanced HTTPS is. We're going to have a look at a site that we will use extensively throughout the rest of the course. So that's what we're going to cover. Let's jump into it and start talking about certificate authorities.

Certificate Authorities

Certificate authorities, or CAs, are an absolutely essential component of secure communication over HTTPS. These are the entities we go to to issue the certificates, which we then load into our website so that our customers can communicate securely. Every time you browse to a website over HTTPS, the owner of that site has used a certificate authority to verify their ownership of the domain and then consequently obtain the certificate that that site can serve up. Let me give you a couple of examples of CAs. Let's begin with my blog and, as you can see, it is served over HTTPS. We've got green padlock, green secure; everything looks good. Let's click on that and drill down and have a look at who issued the certificate; and in this case, we can see that it was issued by Comodo. And in my particular case, the reason why that is is because I use Cloudflare to provide me with HTTPS, and they get their certificates from Comodo. We're going to talk about Cloudflare later on in the course. For now though, that was a really simple way of checking which CA issued me the cert that allows me to communicate securely. Let's go and have a look at another site. This site is have I been pwned?, which is a free project that I run that allows people to see where their email address has been exposed in data breaches. Now, as we can see from the address bar, this site is also served over HTTPS. In this case, we can actually see the name of the project there next to the padlock, which is then followed by my name; and later on in the course we're going to talk about extended validation certificates and how this actually works. For now, let's have a look at which CA I've used for my cert; and in this case, it is DigiCert. So, as with my blog, it's easy just to go and inspect who issued the certificate; and we've now seen a couple of different CAs. The thing is, though, there are significantly more than just a couple of CAs. There are a huge number of them, and the way the certificate authority mechanics work is that your machine needs to trust a CA. The CA signs the certificate; and when it's returned to the browser from the website, your machine validates that the certificate is legitimate by referring to your local list of trusted authorities. Let me show you where to find those. I'm going to press Windows R to run a command, and I'm going to open the application, certmgr.msc; and what that does is gives us our local certificate manager. Now, what I'm interested in here is the trusted root certificate authorities. So we're going to drill down into this, then into the certificates, and what we now see is a fairly extensive list of certificate authorities. Now within those, we can see Comodo, which was the CA that Cloudflare uses for my blog. We can see DigiCert, which I'm using on have I been pwned? And we can see a whole bunch of other names here, some of which may be familiar, many others which are not very familiar at all. The list is actually much more comprehensive than this. Windows will trust many other CAs that it can download as required. So it's not limited to just the certificate authorities that we have listed here. Now this CA list is used by Windows, it's used by Internet Explorer, it's used by Chrome; but it's not used by Firefox. Firefox manages its own list of CAs. Let me show you how to find those. So here's the current version of Firefox. I'm going to jump over to the menu, down into options, over to advanced; and then on the certificates tab here, I'm going to view certificates. And here we see a list of the authorities that Firefox trusts; and as we scroll through these, we're going to see Comodo and we'll also see DigiCert. And as with the Windows CA list, there are many other CAs listed here that Firefox will trust. Now this list can be fluid, as well. Sometimes, new CAs are added; and indeed, there are times where CAs are removed consequently invalidating the certificate that any website using that CA serves up. Let me give you an example of that. In 2011, the Dutch certificate authority known as DigiNotar had a serious lapse in security, which allowed attackers to fraudulently issue certificates for domains they didn't control. Now a really essential component of the way certificate authorities run is that they should only issue certificates to the holder of the domain that it's being issued for, and this is really the very premise of that authenticity component we discussed in the previous module. You need to be sure that when you go to an HTTPS address, you are actually going to the correct website. Now in DigiNotar's case, it later emerged that it was a state-sponsored attack mounted by Iran, who wanted to be able to inspect the traffic of their citizens as they browse sites such as Gmail. It was found that DigiNotar had some serious shortcomings in their security, and consequently they were promptly removed from the certificate authority stores by the likes of Microsoft from their list of certificate authorities in Windows and Firefox, which we just saw. Being a CA carries a huge responsibility, and the consequences of not very carefully carrying out their duties can be severe; and that's simply because of this. When compromises such as DigiNotar occur, this statement no longer holds true. The very fabric of why we have certificate authorities in the first place is torn apart when an attacker can issue certificates for domains that they don't own. So this is why being a CA is such a serious thing. So that's certificate authorities. Let's move on and start talking about SSL and TLS.

SSL and TLS

Let's talk about SSL and TLS because they're both very significant when it comes to talking about HTTPS, and all of these terms tend to be used a little bit interchangeably. For example, people will often say SSL when they mean TLS. Let's talk about what they actually mean though and what the histories are, and then we'll come back to the use of the terminology. So SSL is secure sockets layer, and it was originally built by Netscape back in the early 90s. Version 1.0 never really saw the light of day in public, but version 2.0 hit in 1995; and this was really the first time we started to use a secure transport layer en masse in web browsers. It was later followed by version 3.0 the year after, and that was the last major release of SSL. Now SSL actually lasted a very long time, and it really wasn't until 2014, when we had the POODLE attack that SSL really reached the end of its life. Now that's not to say it wasn't still being used, and I'll come back to that in just a moment. Let's talk about TLS next. TLS is transport layer security, and this came along in 1999; and it was intended to be the successor to SSL, and this is the first really important point about these two terms. Since pretty much the turn of the century, TLS has been the current standard for implementing HTTPS on the web. SSL stopped and TLS began. Version 1.1 came along in 2006, and 1.2 a couple of years after that. As of early 2017, 1.3 still remains in working draft; and with each of these new releases, we're seeing improvements in things like security and speed. They continue to make secure communication stronger, faster, and more efficient. Now the challenge here is that you've always got two parties in the communication. So in the case of HTTPS, you've normally got a browser and a server; and what'll happen is they'll negotiate with each other about which is the strongest implementation they can use. So for example, you might have a server which can implement anywhere up to TLS 1.2. Now if a client comes along and says, well, I can only work as far as TLS 1.1.; then the communication will fall back to the highest commonly supported protocol version. One of the big changes that happened after the POODLE attack was that SSL support starting being removed completely because part of what made POODLE so effective was that attackers could force the downgrade of the communication from TLS to SSL, which had other weaknesses that could be exploited. One final thing on SSL and TLS and the way these terms are used. Because SSL reached so far into our vernacular for so long, people frequently say SSL when they really mean TLS; and it's become a bit of a colloquialism. So for example, if I'm building an asp.net web application and I want to force the application to use a secure connection, I set an attribute called require SSL. Now it doesn't require SSL at all; it requires HTTPS, which these days is going to be implemented using TLS. Yet it's still called SSL. We're going to look at a website at the end of this module that is enormously useful for testing TLS, which is called bad SSL. So on the one hand, it's important to be conscious that these two things are different; but on the other hand, you also need to be aware that the terms are used interchangeably. With that said, let's move on and have a look at the mechanics of how a TLS handshake actually works.

The TLS Handshake

When a client, such as a browser, wants to connect to a server over HTTPS, they begin what's known as a TLS handshake. So this is where the client and the server need to negotiate with each other and agree on how they're going to communicate securely. Now this handshake consists of a client hello, and within this request, the client communicates things such as the highest level of TLS it will support and other information such as supported cipher suites in order of the preference that it would like to use. The server will then respond with a server hello; and in that response it will agree on the protocol version and the cipher suite, as well as providing its public key back to the client. Now the client can then verify that public key against its list of certificate authorities, and this is where we get the confidence that we are actually talking to who we think we are. Now one thing that's important to note here is that this initial communication is not yet encrypted. This is just the negotiation phase. So a man in the middle could see that the client is trying to talk to the server, and it could know the identities of both. There's not contents to the communication yet; it's just trying to negotiate that initial connection. The client can now perform a key exchange with the server, and this response is encrypted with the server's public key; to which the server can now return a server finished response; and the secure communication can begin. Now, obviously, there are a bunch of mechanics within that exchange; but from the perspective of what you must know in order to build applications on HTTPS, that doesn't matter too much. The important thing to understand is that there is a negotiation, you can see which client and server are talking to each other, and that once the communication is securely established, then the contents of the requests and responses become encrypted. So that's the TLS handshake. Let's go and have a look at issuing certificates for local development.

Issuing Certs for Development

When you're building applications locally and you want to test them over HTTPS, you're going to need a certificate; but you're not going to go and get one from a certificate authority. This is just local development. We just have to be able to create an HTTPS connection, and it doesn't matter whether it's a valid certificate authority or not. Now this is easy to do with PowerShell. If you're on a platform other than Windows, have a look at the likes of open SSL to do the same thing. Now in this case, what I'm going to do is create a certificate for the DNS name of example.troyhunt.com. So I'm going to run that now. Now one important point to make here is that you can see I'm running PowerShell in administrator mode. You will need to run under a privileged account in order to create a new self-signed certificate. The certificate is now created. We've got a unique thumbprint, and we can see the subject here is that host name that I just created. What I'm going to do next is copy this thumbprint because I'm going to need to use that in a moment to export this certificate from my local certificate store. Ultimately, what I want to do in this example is create a PFX formatted certificate; and in order to do that, I'm going to need a password that I can set on the certificate. So let me now paste in another command, and this is going to take the password we see here, the one starting with three-zed-N, and convert it to a secure string because I'm going to need that for the next command. So let me run that and now drop in the final command to export my PFX. Note here that I have the thumbprint of the certificate from earlier on. So this export PFX certificate command needs to know which certificate to export based on the particular thumbprint. I'm going to export that into the cert folder on my C drive and call it cert.pfx; let's run that. And that certificate has now been exported. Let's have a look at it in Windows Explorer, and there is my self-signed certificate. I can now install that into IIS or NGINX or whatever other platform I want to run on where I'm writing and testing my applications. This is certainly not the only way to create a local, self-signed certificate; but I wanted to show you this model because it's probably about the easiest if you're working in a Windows environment. Next up, let's have a look at how we can intercept HTTPS traffic during debugging.

Intercepting HTTPS Traffic During Debugging

Many people use Fiddler quite extensively for debugging web applications, and Fiddler works great straight out of the box for HTTP; but let's have a look at what happens if we want to capture HTTPS traffic. And to demonstrate that, I'm going to drag over a browser window and, with Fiddler still sitting behind us here, let's try and load a website over HTTP. So for example, a website such as cnn.com. Now, I chose cnn.com because I know that it loads over HTTP; and we can see a whole bunch of HTTP requests appear in Fiddler. That's precisely what we'd expect. However, let's now go and change CNN to, say, my blog instead; and we see no new requests appear in Fiddler because this is an HTTPS connection that's being encrypted in my browser before it goes through Fiddler, which is a proxy, and before it actually then gets passed on to my website. Now to fix that, I'm going to jump over here into tools, down into Telerik Fiddler Options, over to HTTPS; and now we have a couple of options. I'm going to begin by just capturing HTTPS connects, and you can see here that this intercepts HTTPS and web socket tunneling requests. Now this is the beginning of the negotiation phase that I mentioned in the clip on the TLS handshake. Let's start by just checking that box. Okay to that, and now bring my blog back up and I'll go to a new website. So let's say the New York Times, which I mentioned in the first module is now served over HTTPS. If we drop back to Fiddler now, we can see that in loading the New York Times, we had a request here that's referred to as tunnel. So this is the beginning of that negotiation, and we can see the host name that was requested here; and we can see that the request went over port 443, which is the default port we use for HTTPS. Now towards the top right of the screen, we can see that HTTPS decryption is disabled; and we can click to configure. Alternatively, we can go back to tools, Telerik Fiddler Options, over to HTTPS, and check decrypt HTTPS traffic. Now when we do that, we get a prompt here that literally says "scary text ahead"; and it goes on to explain that Fiddler is going to generate its own unique root certificate. This is then going to load that certificate into the list of trusted CAs that we saw earlier on. The reason this is scary text is that it is going to allow Fiddler to intercept a secure communication. Only on my machine, of course, because that's where it's installing the certificate; but it is a significant thing regardless. Let's trust that certificate, and we'll agree to this second security warning, which talks about how the certificate is going to be called Do Not Trust Fiddler Root. We'll see that one again in a moment. Let's say yes to that. Then you'll see a Windows user access control prompt followed by one final prompt that we'll say yes to, and finally Fiddler's root certificate has been added. So let's dismiss those dialogues, and I'm going to restart Fiddler. Now I'm going to grab another browser window; but before I even do that, we can see a number of requests already coming through over HTTPS. I've got another browser window open on another screen, and Twitter's open so we can see Twitter requests. I've got a tab with my Azure management portal open, so we're saying request to that. Let me bring that other window back over, and we'll try loading my blog again. Now because the front page might be cached, I'm going to give it a hard refresh; and then now we can see that request zooming by on the screen because there are now many requests being issued through Fiddler, and here is the full HTTPS request intercepted by Fiddler. Now what Fiddler has done is it has mounted a man in the middle attack on my own connection and issued its own certificate to my browser; and what that means is if I jump back to my browser and then go and inspect the certificate, we now see that the CA is do not trust Fiddler root, which is what we just saw on one of those previous prompts. So what we've really done here is compromised my machine such that it trusts an invalid certificate authority. Now this doesn't put me at any further risk. It's only my machine which can intercept my own requests, but what it allows me to do is inspect HTTPS communication in just the same way as I could do with HTTP. So this gets us up and running with intercepting secure requests for the purposes of debugging. Speaking of debugging, let's move on and look at one more resource before we wrap up the module.

Using badssl.com

This is badssl.com, and it allows you to test bad implementations of TLS; and this speaks to the point earlier on about the interchangeability of the terms SSL and TLS. This is a fantastic site because it illustrates all the things that can go wrong with HTTPS and the way it's configured within applications. So for example, what if you make an HTTPS request to a website that has an expired certificate? Let's see what that looks like. And Chrome gives us a very overt warning, which is good. This is exactly what we expect. We can go and have a look at what it actually served us up certificate-wise; and this certificate expired in April 2015, so well and truly out of date at the time of recording. This is perfect for seeing how a client behaves when an expired is served. It also gives us the opportunity to see how different clients behave in different ways. So for example, let me go and grab Microsoft edge; and we'll jump over to badssl.com again, and let's have look at an expired certificate. Another massive warning; but, of course, somewhat different to the one in Chrome because it's a different client. It's representing the information slightly differently. There are many, many different examples of improperly implemented HTTPS on badssl.com; and as we proceed through this course, particularly in the next module, we're going to use badssl.com in order to experience what those behaviors look like. For now though, that's it for this module. Let's go and summarize what we've just covered.

Summary

Let's summarize the module, and the first thing we looked at was certificate of authorities; and, clearly, they are an essential part of the HTTPS communication. We rely on them to generate the certificates which our websites then return to client, and we rely on them to only create those certificates for valid account holders of the domain. Otherwise we have situation like DigiNotar. We looked at SSL and TLS, which are not the same things, nor are they the same as HTTPS. They are different things; however, the terms are regularly used in an interchangeable fashion. We took a look at Fiddler and the ability to issue a root CA for the purposes of debugging. This is enormously useful when you're developing HTTPS websites, and it was also a good demonstration of how certificate authorities work in terms of adding their root CA into the local trust store. And, finally, we just looked at badssl.com; and we used it to test some bad TLS, which again demonstrates the interchangeability of these terms. We'll keep coming back to bad SSL because it's an enormously useful demonstration of how browsers behave. So that's it for the module on HTTPS fundamentals. Next up, let's go and start to look at how we're actually going to secure our applications.

Securing the Application

Overview

In this module, we're going to start looking at what it takes to actually secure the application. So we're going to start looking at the code level, so now that we know why HTTPS, is a good thing, and we understand some of the fundamentals, let's actually start looking at the code level. So let's look at what we have to do with our HTML, in order to implement HTTPS as securely as we can. This is a really important module, probably the most important module of the course. So let's go and take a look at the overview, and then we'll jump right into it. A lot of people think that implementing HTTPS is a basic thing, and certainly, it can be easy. But there are some really important aspects to how we implement it, and there's more to it than just simply redirecting insecure requests. I'm going to show you why in just a moment. Now properly securing a site with HTTPS, can have a pretty broad reaching impact, and what I mean by that, is that particularly for existing applications, so Brownfield applications, legacy applications, there can be site-wide work which needs to be done, depending on the design of the site, you may need to make changes to every single page. But there are also constructs that can help us go secure on a site-wide basis, and we're going to have a look at some of the controls, that can help you do that. And finally, the implementation of HTTPS, can be a bit nuanced, so there are lots of little tricks that we want to try and get right. And when we combine all of these things together, it can help make the move to HTTPS, that much easier, and result in a much more secure implementation. So that's enough of the overview, let's go and start looking at how to secure the application.

Redirecting from HTTP to HTTPS

Let's start with one of the most fundamental things you're going to want to do, when moving a website to HTTPS, and that is to actually get it to load, over HTTPS. Let me show you what I mean, this website, The World's Greatest Azure Demo, is a website that I built back in 2014, which serves a pretty self-explanatory purpose. I built this in order to demonstrate Azure, lots of different features of Azure, and I later also turned into a Pluralsight Course, called Modernizing Your Websites With Azure Platform as a Service. Now when I originally buildtthis site, I didn't see a need for HTTPS, it's just a basic, informational site. But for all the reasons I've already outlined earlier on in this course, now is the time for it to go to HTTPS. And so that's what I've done, I've put this particular site behind Cloudflare, which we'll talk more about later on in the course, such that, I can now change the URL to request it over the secure scheme. But of course, to be secure, this is what we want to do by default. I don't want people coming to the insecure version of the site, so what I want to do is redirect the traffic, from HTTP, to HTTPS. Let me give you a really basic overview of what I mean. Let's talk about this in really simple terms, we have a client, like my browser, and it makes an HTTP request to the server. Now that might be because someone is explicitly going to an HTTP address, so they've clicked a link from somewhere, or they've actually typed in HTTP, colon, forward slash, forward slash. Alternatively, it may be because they've just typed in the host name, they've typed in WorldsGreatestAzureDemo.com, into the browser, and all browsers, presently default to the insecure scheme. So this is one of the problems we have with securing the web, and we'll talk more about how we can defend against that risk as we go through this module. Regardless of how it happens, the server receives that insecure request, and it's now going to respond with an HTTP 301. Now a 301 is a permanent redirect, it's going to tell the browser, that you need to know go and make a subsequent HTTPS request back to the server. We'll have a look at the network traffic in a moment of how this works, and you'll see that, that HTTP 301 response, includes a location header, with the secure scheme in it. So this is how the browser knows to go back and make another request. I'm now going to go and implement that, in The World's Greatest Azure Website, and then we'll see how the behavior changes. So I've just gone and enabled the HTTP 301 redirects for this site, which means that, any insecure requests that the server receives, it should now turn around and redirect the browser to make a secure request. Let's see how that works, but before I reload this page, I'm going to hit F12, to open up the Dev Tools, I'm on the Network Tab, I want to show you what the traffic does. So let's do this, let's go up, and reload the page. Now have a look at what's happened here, we'll scroll back up through the requests, and we'll see two requests at the top, that are quite important. The first one has a status of 301, let's draw down and look at that request. And we can see here, in this General Section, the requested URL was the insecure scheme, the status code is 301, and if we look at the very bottom of the screen, we can see that the response headers have a location value. And it's exactly the same as the URL we requested, except for the S in the scheme. Now the mechanics of how this works, is that the browser has received this request, and then gone and made a subsequent request, which is this one just here. Now this request has gone out securely, and we've got an HTTP status of 200 that's come back. So this is a good request. So the first thing to point out here, is that this is how the 301 works, it results in the browser making two requests. And if we close the details of that request, we can see over on the right hand side of the screen, the little waterfall diagram. Where we've had to go out to make a request, that's taken 433 milliseconds, received the response, and then gone back out, and made another request, which has taken another quarter of a second, in order to get the secure page. Now as it relates to security, there's one major problem with this, and that is, that the first request here is insecure. Yes, the server has redirected the browser, and caused it to subsequently make a secure request, but that first request is at risk. And the same thing will happen every time we do this. Let's remove the S, and try it again. And here it is again, the unsecure first request, resulting in the 301, and then the secure request. So you can see the problem here, if an attacker can get in the middle of the traffic, they can always intercept that first request. They could sit there and proxy the traffic backwards and forwards securely to the server, and then backwards and forwards insecurely to the user. Or they could get that first request, and just redirect the user to somewhere else, they could DNS spoof that first request, and return their own content. Clearly, we have a problem with this basic model of just simply redirecting the first request. Fortunately, we have a solution, and that solution is HTTP Strict Transport Security. Let's go and take a look at how that works now.

HTTP Strict Transport Security (HSTS)

We just saw how an HTTP 301 can be used to redirect requests made insecurely, to requests made over HTTPS. Now that was great, if you could get that first request past that man in the middle. But of course, that does also leave a problem. Now the solution to this is HTTP Strict Transport Security, and you'll often see this abbreviated to either STS, for Strict Transport Security, or HSTS, for HTTP Strict Transport Security. I've just enabled HSTS on this website, and what I want to do now is reload this page, keeping in the mind this is the secure page, we're going to reload it, and we're going to have a look at the response headers of this site. Let's give a go. Now if I scroll back up to the first request, draw down on it, and then scroll down in the response headers, we can now see a value for Strict Transport Security. This is the HSTS header, and you can see here that it's got a max age value, and this is expressed in seconds, this particular value is about one month's worth of seconds. Now I'm going to explain why the max age value is significant in a moment, before I do that though, let me show you what this actually does. I'm going to go back up to the address bar, and attempt to load this page over HTTP. Let's see what happens this time. We now see a similar, but slightly different pattern, we see two different requests to WorldsGreatestAzureDemo.com, but whereas before, the first request resulted in a 301 status. Now we have a 307, not only do we have a 307, but look at the size of it, it's zero bytes. It also, it only took six milliseconds to load, and that is much faster than what it takes my browser to make a request to this website. What the browser has actually done here, and we see it as we hover over the status, is it's performed a 307 internal redirect. Now if we drilled down on that request, we can see that 307 represented in the general category, and then down in the response headers, it says, the non-authoritative reason is HSTS, and then of course, the next request down, is the secure request. Now let's go and have a look at that response header again, for Strict Transport Security. The way it works, is that once your browser sees the STS response header, for the period of time, specified in that max age value, it will not make an insecure request to that domain. So this is great, because what we've just done, is locked the browser into only making secure requests for another month. If I close my browser, go away for a couple of weeks, and then come back, and try to load this site again, even if I just type in WorldsGreatestAzureDemo.com, with no scheme, it defaults to the insecure scheme, the browser will 307 it, before it even goes out over the wire. So I'm protected, and that's great. However, there is one problem remaining, and it's this, TOFU, Trust On First Use. That HSTS response header works fantastically, but we have to get it first, and what that often means is an insecure request going out first. So consider the life cycle, someone tries to go to this website, they type in WorldsGreatestAzureDemo.com, into the browser, the browser makes a non-secure request first, over HTTP, then it receives a 301 response, now it goes out and makes a secure request, and then it gets the HSTS header. That header can only be returned over a secure request, but even if you could, we've still got the same problem of the first request being unsecure. And that's this whole TOFU paradigm, we've got to trust just one good request. However, there's a fix for this, I'm going to go and make another little change to the application, and then I'm going to show you it works.

Preloading HSTS

Okay, I have just made some tweaks to the way this application is returning the HSTS header. Let's not give it a reload, and we'll have a look at what's different. And we'll go and grab that first request up here, down to the HSTS header, and now we can see a few little changes. So the first thing is, my max age is much longer, this is now one year's worth of seconds. It's no longer a month, you'll see why that's important soon. I've also added this include subdomains directive, and this does exactly what it sounds like it does. It's going to make sure that any subdomain of WorldsGreatestAzureDemo.com, also has the same HSTS policy applied to it. And finally, and this is the bit that's going to fix our TOFU problem. I've got this preload keyword as well, now to understand what preload does, we need to go and check out HSTSPreload.org. This site is run by the Chromium Project, and the information here on the screen, actually does a pretty good job of explaining what it's for. What the HSTS preload site does, is allows you to add your website to the list of sites that are hard coded into Chrome, as being HTTPS only. Not just Chrome either, it works with Firefox, Opera, Safari, Incident, Explorer and Edge, these all have HSTS preload lists, based on the submissions that are made to this site. Let's go and put in that WorldsGreatestAzureDemo.com domain, and go through the process. And then we'll talk about some the prerequisites as well. The site is not presently preloaded, however, the site is eligible for preloading, now I'm going to confirm that I am the owner of the site. And I'm also going to confirm that both the site and all subdomains will no longer be accessible without a valid HTTPS certificate. Now just before I submit that, let's scroll down and have a look at the submission requirements. And this will help you understand why I've configured the site as I have. So in order be accepted to the HSTS preload list, you must serve a valid certificate, which is fair enough. Redirect all your traffic from HTTP to HTTPS, we just did that with the 301. Serve all subdomains over HTTPS, now that's why I needed to include the subdomains keyword. Number 4, you've got to serve an HSTS header, the max age has got be at least 18 weeks, so that's why I changed it from one month to one year. We've covered the include subdomains directive, the preload directive has got to be there. So this is the one I just added, and that's what makes this site eligible for preloading. You can't go and preload someone else's site, and force it to only ever load over HTTPS, the site has got to actively return that preload directive. And then if you're going to redirect on the HTTPS site, you've still got to have an HSTS header on that redirect as well. And this is it, we can now scroll back up here, and submit this website for preloading. Great, it is now pending inclusion in the HSTS preload list. What this means, is that once the browser vendors pick up WorldsGreatestAzureDemo.com from this list, and then they build it into their browsers, anyone using a browser with this preloaded, and remember, it's preloaded right out of the box, right from the moment the manufacturer sends it. No one will be able to load this site insecurely. This solves the TOFO problem, because you don't need to do any trust on first use, there simply will not be an insecure connection in the first place. Now this also means you can't go back, you can't suddenly decide that you no longer want to serve either the primary website, or a subdomain, over HTTPS. This is a one way street, and if you change your mind later on, life is going to get very difficult. So that's great. There are two more things I want to show you about HSTS, and the first one is this, this is Chromium's list of all preloaded websites, and if we have scroll through them, we can see that there are thousands, and thousands of different sites that have been preloaded into the browser. In fact, if I do a search for the project I run, haveibeenpwned, we'll see that, that's here, that is preloaded, you cannot make an insecure request, to my data breach notification service, it's preloaded into every browser, and it's been there for quite a while too. So this is a really good resource to go and have a look at who has been preloaded where, and you'll see a list of every website that is presently preloaded into the browser. Let's look at one final thing, before we move on, and that's compatibility. So over on CanIUse.com, and having a look at the HSTS compatibility, it's actually pretty good. When we look at the global stats, over 83% of traffic is using a browser that understands HSTS. We're a bit further ahead, down here in Australia, more than 95% of our traffic. And when you look at the browser support, further down the screen, you can see why, every major browser presently in circulation, supports HSTS, and it's a great way of saying, we're going to go HTTPS forever, on our website. So this is great, between the HTTP 301, which you have to have to preload, and the HSTS header with a long max age, we include subdomains, and the preload directives, we've got a really solid implementation for forcing requests to go HTTPS. This is a great foundation. Let's now move on and have a look at some of the changes that we're going to have to make to the site, in order to implement HTTPS properly.

Understanding Mixed Content

Now that we're redirecting all traffic on this website to HTTPS, both via the 301 redirect, and via HSTS, it's time to actually start securing the site properly, because we've got a couple of problems here. Now the first problem is that, when we look in the address bar, and we look at the HTTPS scheme, it's not green, and there's no padlock. And this is telling us that there's a problem with our implementation. Now when you're working on configuring HTTPS on a website, it's always quite useful to look at the Console, so I'm going to jump down the Dev Tools, and then over to the Console Tab. Now we can see we've got both a warning and an error here, and we're going to solve these one by one, and talk about why they're appearing.

Let's start with that warning, it says, there's mixed content, the page was loaded over HTTPS, but it requested an insecure image, and we can see here that the image is actually the Pluralsight logo. Now we can click down there, and have a look at the image, which is this one just here, and if we scroll down a little bit on the panel above, we can see that image appearing on the page. So the problem is, this image is being requested insecurely, we can see it here in the Element Inspector, where it's being referenced explicitly as HTTP colon, forward slash, forward slash, now how to think about what this means, even though we can trust the HTML page, so we know that no one has been monitoring that traffic, modifying the traffic, or redirecting it to somewhere else, we now can't trust the image. Somebody could have changed this image, now interestingly, if we go and right click on that URL, and open it up in a new tab, we see the secure image load here, and this is because we've got HSTS enabled now, so the browser wouldn't actually let me make an insecure request for it. But the problem is that it is embedded insecurely, and that's what we're going to fix now. I'm going to go off and fix that reference, then we'll come back and reload the page. Okay, I've just gone and fixed the reference to this image in the source code, let's reload the page and see how it goes now. Fantastic, I have now got green in my address bar, green padlock, green secure, green HTTPS. And if we go and inspect the image, you can see that all I've done is just change the path to be relative. Rather than explicitly specifying that it's TheWorldsGreatestAzureDemo.com, and explicitly referencing the scheme, I've just said, hey, grab it from the forward slash content, forward slash images, forward slash Pluralsightlogo.svg path. And that request will be made with the same secure protocol that the parent page is. So this has fixed our problem, we've now gone green in all the places, and it was a simple fix. You need to keep this in mind, because anywhere in your application where you embed images like this, over an insecure connection, you're going to have the same challenge. But we're not clear yet, because there was one other error in the Console, and if we go back there now, we can see where the problem is. Now this error sounds a little bit similar to the previous warning, in that it says, the page was loaded securely, but then it requested an insecure resource. Now this is curious, because we've got the green padlock, and the green secure, and HTTPS now, yet we've got this error. Now the reason why is that this is a subtly different problem. And it's to do with active versus passive content, when we loaded the image, the image was passive content. Now what that means is that, it doesn't actively do anything on the page. It just sits there, it's an image, it can't perform any functionality, and as a result, the browser still loaded it. It said, hey, this might be bad, but it can't actually do much, so we're going to load it anyway, we're just going to take away your green stuff in the address bar. Now this is quite different, because this is trying to embed YouTube, and when you embed YouTube, you embed an iframe, an iframe can have active content in it. So you have things like scripts, and scripts can actually do a lot of damage, so the browser is doing a risk assessment on the likelihood of the content doing bad things. Now what this means is that if we close the Dev Tools, and scroll down the page, right about here, there would normally be a video. And instead, we've got a great big, blank space. Not only that, but there is actually one other visual indicator on the page, and that's this shield up in the top right hand corner. Let's have a look at what it says. This page is trying to load scripts from unauthenticated sources, and what's actually happened is, it's blocked it. Now we can go and load those unsafe scripts, let's have a look at what happens then. Now we see my video, and not only do we see the video, but now we've got some very overt visual warnings in the top left of the page. We've gone from green everything, through to a red alert symbol, not secure text, and a big cross through our HTTPS. This is not a good look, and it's the browser's way of saying, hey, you've loaded content insecurely, which could actually by quite damaging. Now if we go and inspect the element where this YouTube video is embedded, we can see why that is. So there's the iframe, and you can clearly see that the source of it is an insecure address. Now fortunately, this is also a really easy fix, let me go and do that, and then we'll come back here and reload the page. Okay, I have just fixed my site, so it loads this YouTube video securely. Let me refresh the page, and we'll see what happens. And now the page has loaded without any visible errors, so we've got green all the way through the address bar, no warning symbol in the top right, and our video has loaded too. Let's just check the Dev Tools and make sure there's no warnings in the Console, the Console looks clean. So now, our page is good. Now let me show you how I did this, because I've used a slightly different approach. Let me go through and inspect the element, and what you'll see this time, is that I have omitted the scheme altogether. In fact, the source attribute of this iframe, is using what's referred to as a protocol relative path. So no HTTPS colon, forward slash, forward slash, just starting with the forward slash, forward slash, now that will load this embedded asset over the same scheme as the parent page. So if I did go back and request this page over HTTP, and it didn't redirect, and there was no HSTS, the YouTube video would be loaded over the unsecure connection. And then, obviously, when I load the page over the secure connection, this embedded resource inherits the same security profile. So protocol relative URLs kind of give you the best of both worlds, if you're transitioning to HTTPS. Frankly though, as we move forward into an HTTPS only world, you might as well just have these references explicitly as HTTPS, colon, forward slash, forward slash. You're really not left with many cases where you'd want to embed that content insecurely. So we've just looked at the necessity to ensure that everything from images to iframes, and of course, anything else embedded on the page, C-Assist, Java Script, fonts, anything you can think of, we've just looked at the necessity to ensure that all of those are loaded securely. Now this can be a laborious task, because you're going to have make sure every single embedded resource on your site, doesn't pull that content insecurely. But there's another way of doing this, a way that can make your life a lot easier. Let's go and have a look at what that is.

Using a CSP to Upgrade or Block Insecure Requests

We just saw how your site needs to ensure that every single resource that is embedded into it, is served over a secure connection, otherwise, you get the errors that we just saw. Now this can be laborious, going through you whole site, trying to find all of these errors and clean them up, and you do need to do it. What I'm about to show you, doesn't mean that you can skip that, but there is a way of streamlining this processes, and giving a little bit of insurance, should you have missed anything. I've just reconfigured the site, to load the Pluralsight logo from the same site, insecurely, so we've lost our green bits up there in the address bar, and also embed the YouTube video insecurely in the iframe. So we've consequently got the shield, and the red cross in the top right of the screen. I'm now going to fix all of this in one go, with one simple tag, let me go and do that, and then we'll come back and see how it works. Okay, I've just made my change, now let's give the page a refresh, and have a look at what happens. And this looks pretty good, we're back to green bits, and our broken shield is gone. If I click on the link to watch the demo, the YouTube video is now rendering onto the screen. So all of this is now secure, however, let's have a look at the Element Inspector. And what you'll see here, is that, the video is still embedded insecurely. So how come the page now says we're secure, even though the video is embedded insecurely? Well the way I've done this is, via a meta tag, and if we scroll up, and have a look at the head of the document, you can see I've got this content security policy meta tag, and the content of it says, upgrade insecure requests. Now this is a CSP, which is content security policy, and I've got an entire Pluralsight course, on Introduction to Browser Security Headers, which explains how to implement content security policies. Now I just said browser security headers, and you're seeing a meta tag here, you can add a content security policy, either via meta tag, or via response header. And I thought I'd show you the meta tag here, because it's probably the easiest way for most people to add this. Now what this does, is exactly what it sounds like it should do, which is that every single insecure request, embedded into your page, gets upgraded to a secure one. One little meta tag, or response header, and it fixes everything, mostly. Here's the catch, browser support. Now it's not completely terrible, globally, we do have 62% of present browsers understanding the upgrade insecure request CSP. In this case, Australia is a little bit further behind the rest of the world. But as you can see by looking at the browser support, there's no support in Internet Explorer, no support in Edge, and no support in the current version of Safari. So really, we are pretty limited with support, now what that means is that, if you just go and implement this content security policy, and you don't actually fix the references in your HTML, yes, more than half your audience will now get content served securely, and that's great. But a large number of your visitors, are going to have problems, they're going to see browser warnings. They're going to be left at risk, of a man in the middle attack. So the best possible thing to do is both of these things. Go through your site, and fix every single insecure reference you can find, but also use the upgrade insecure request CSP. Because what that means, is that if you did miss something somewhere, there's a better than average chance that when that page loads, the user will be on a browser that upgrades the request for them. That's a great transitional state, but there is also a very closely related content security policy you can use, as you progress in your HTTPS implementation. Let me show you what this one does. Have a look at the site now, and we'll start from the top of the screen and work down. We've got green everywhere, which is good, we've got no warning in the top right hand corner of the browser, which is also good. However, we also have no Pluralsight logo, we can see the alt text sitting there on the screen. And if we scroll down to where the video would normally be embedded, that's not there either. If we jump into the Console, and we have a look at the errors, we can see a couple of them related to HTTPS. And they both talk about mixed content, and they say the page was loaded over HTTPS, but requested an insecure resource, and it's been blocked. I'll show you how that blocking works, I'm going to go up to the Element Inspector, into the head of the document, and then here's another CSP. And this one is block all mixed content, so it's related to upgrade insecure requests, except it does exactly what it sounds like it should do, it blocks it altogether. This is a much more nuclear approach, so this is the one that says, if I haven't got my site exactly right, it's going to block the content altogether, not just the active content, like iframes and scripts, but the passive content as well, like images. This is the safest possible position to fall back to, because it's the one that assumes that you have got all your code right on the site, and nothing else will load, except what you have explicitly told the browser to load over the schemes that you've told it to. This is another step in the evolution, and I use this on sites where I'm really confident I've got everything right. As you're transitioning, use the upgrade insecure requests, once you're confident it's all secure, then block anything that is not outright. So that's how to use those CSPs, now let's go on and have a look at securing our cookies.

Secure Cookies

The security of cookies that a website sends backwards and forwards to the browser, is really important, because cookies can contain enormously valuable information. So for example, cookies are used for persisting authentication tokens, so we often refer to auth cookies. If you managed to obtain an auth cookie, then you have the ability to hijack someone's session, and if that's a foreign concept to you, go and check out my Pluralsight course, on Ethical Hacking, Session Hijacking. It's a very, very damaging attack. Now I've just gone and set a couple of cookies on this site, and I want to show you what they are, so if I hit F12 to drop into the Developer Tools, then over to Application, and we're already down here, in the Cookies Section, and we can see there are number of cookies that are presently set for this site. The two that I've just set, in order to do this demonstration, is the insecure cookie, and that's the name of the cookie, and the secure cookie. And if we have a look at these two rows, they're very similar, they're valid for the same domain and path, they both expire at the end of the session. The only real difference is that the insecure cookie doesn't have a tick in the Secure Column, but the secure cookie does. Now let me show you what this means. I'm going to go over to the Network Tab, reload the page, and then we're going to go and take a look at the very first request. Down to the request headers, and we can see both the secure cookie, and the insecure cookie have been sent. Now we're looking at it in the browser, but you were to look at it on the wire as well, so for example, if you were to proxy your browser through Fiddler, and look at the traffic, you'd see that both of those cookies just went. Now let's do this, I've just disabled the HTTP 301, that we set earlier, and I've also disabled HSTS, it wouldn't have been preloaded into the browser yet, so I've actually gone and deleted this site from Chrome's net internal setting. Now what that means, is that I can now go back and make an insecure request to the site, let's do that. Now let's go and inspect this request. Down to the request headers, and now, we can only see the insecure cookie, the secure cookie hasn't been sent, because when it has that secure flag on it, it will not be sent over an insecure request. Now this is really important, because any cookie, of any value whatsoever, should be flagged as secure, so it can't accidentally be leaked over an insecure request. In fact, I would even go so far as to say that, every single cookie should always be secure, unless there is a very explicit reason that you'd want to return it over an insecure connection. Now in some ways, this is a little bit redundant, because if you're using HSTS, then the browser won't send it over an insecure connection anyway, especially if your HSTS is preloaded. However, you've always got the TOFU problem, if it's not preloaded, the fact that are still some browsers that don't support HSTS. And then there's the fact that we always like to do defense in-depth, so having multiple layers of security, one on top of the other, sometimes seemingly redundant, such that, if one of them goes wrong, you've got other fallback positions that will save you from a malicious attack. So secure cookies are really important, default to that position, and anything else is an exception. So that's insecure cookies. And that rounds out the fundamental aspects of securing the application. Let's go on and summarize the module.

Summary

During this module, we've seen that, in order to implement HTTPS, it does involve multiple code changes. So for example, your references to embedded assets, might need to change, that includes both for passive content, like images, and active content, like iframes and scripts. Not only that, but we need to redirect insecure requests, to secure ones. And then there were the various response headers and meta tags we looked at. So by the time you're done, there is actually quite a bit you need to add, in order to do HTTPS properly. Now, if you don't do your HTTPS implementation properly, the browsers will get very upset. We've been looking at Chrome, but all the other major browsers, throw warnings of a similar nature. If you have antipatterns, such as mixed content on the page, the browsers are going to get upset, your users are going to get worried, and some stuff, like the video on that The World's Greatest Azure Demo page, may just simply not work. Now obviously, in that case, we just had to go and make sure that, that content was being embedded securely. But we also went through things like HTTP Strict Transport Security, and content security policies, in order to help make that transition easier. If we use a combination of HSTS and CSP directives, such as upgrade insecure requests, it can both make life easier to do the migration, and provide a fallback position, if there's content you've missed, which would otherwise still load insecurely, and cause the sorts of problems we saw during this module. Now doing that, so having multiple different means of loading content securely, is also a great defense in-depth practice. And as I mentioned just a little while ago, this is the whole premise of, let's not just do one thing to secure ourselves, let's try and layer it, let's have multiple fallback positions for if things go wrong. So using attributes such as a secure flag on your cookies, that should be totally redundant, if you get your HSTS right, and you get it preloaded as well. But redundancy is good, this is what we want, so applying defense in-depth as you go through your upgrade to HTTPS, is really important. So that's everything I wanted to cover in the module on Securing Your Application. We're going to look at some more detail, a little bit later on in the course, but first of all, let's move on to overcoming some of the perceived barriers of migrating to HTTPS.

Overcoming (Perceived) Barriers to HTTPS

Overview

In this module, we're going to drill right down into overcoming the barriers to HTTPS, and as per the title here on the screen, these are perceived barriers. There are very, very few good reasons left today not to use HTTPS on everything, and I'm going to use this module to tackle each one of these assumptions that people have traditionally made about why HTTPS isn't suitable and talk about how to overcome those barriers. Let's jump into it and take a look at the overview. There are a number of key reasons you keep hearing from people about why HTTPS won't work, and they've clung to these excuses for many, many years. But the environment around them has changed, and many of the traditional barriers that people believed would stop them from properly securing their traffic simply no longer exist. In fact, it goes even further than that because as we'll see in this module, there are aspects of HTTPS which can actually make your website better than not having it at all. And I don't just mean in terms of security. In fact, I want to focus on one particular aspect of HTTPS, which can be enormously helpful for websites today, and it's got nothing at all to do with encrypting your traffic. I love showing this demo because many people find it very surprising. You'll see that in full swing when I talk about making HTTPS go fast. So this is a really important module because it's the one that should help you get past those traditional barriers to implementing HTTPS across everything. Let's jump in and start by looking at the impact of HTTPS on server performance.

The Impact of HTTPS on Server Performance

This is a great place to start talking about the impact of HTTPS on server performance. Because it's a very self-explanatory URL, istlsfastyet.com. And as you can see from the heading here, this site is rather enthusiastic about the positive speed attributes of TLS. The first paragraph here talks about many of the things we've already discussed throughout this course, so the reasons why HTTPS is important. Now, I'm not going to go through this site in detail, and it does get into detail. So this is a really good resource worth actually reading up on later on. But there's one particular aspect of this site which I do want to call out, and it relates to how Google went HTTPS by default for Gmail back in 2010, and I want to show you a couple of quotes on that. I want to start here, and this is a quote by Adam Langley, who's a software engineer at Google. And there's some really important observations he's made here. And best of all, these are quantifiable observations. So he said when Google went HTTPS only for Gmail, which of course as we know by now, is a pretty essential attribute of an email platform, Google actually found that the SSL or TLS implementation, and we were still using SSL back in 2010 as well as TLS. He reported here that Google found that that encryption implementation accounted for less than 1% of CPU load. That is a very, very small amount of the overhead that's placed on machines running web applications like Gmail. He goes on, less than 10 kilobytes of memory per connection and less than 2% of network overhead. Of course, there has to be some network overhead because there's that negotiation phase that we saw earlier on. But per Adam's observation here, it's very, very small. One other point he makes and this speaks to the perceived barriers of HTTPS, is that he's hoping these numbers will actually help dispel the myth that SSL or TLS or HTTPS actually has any sort of significant overhead on infrastructure. And this was all the way back in 2010. We've moved on a long way since then. So that's the first thing I want to touch on. Nice hard tangible numbers about the impact of secure connections on the server. Let me go and show you something very different, and this is actually quite topical at the time of recording. One of the world's largest adult websites, RedTube, has just gone HTTPS only. Now this is not just one of the largest websites of this particular nature, it's one of the largest websites full stop. It's up there within the top couple of 100 of all websites used all across the internet. Earlier on in the course, I pointed out the likes of The New York Times just going HTTPS, and that was an important observation because a website like that doesn't really serve any content where confidentiality is important. RedTube is very different for a couple of reasons. And one of the reasons, and this is particularly relevant to server performance, is that there is a huge amount of streaming media. Now as easy as it can be to implement HTTPS on most normal websites, once you start dealing with large volumes of streaming media, now things do start to get harder. As you can imagine, being one of the top couple of 100 websites in the world and streaming high definition media to a huge global audience is going to put a lot of demands on infrastructure. But that's a challenge that these guys have been able to overcome. And as I said, this is only just happened at the time of recording. So clearly, like Gmail did all those years ago, they've managed to overcome that server overhead in order to serve traffic to their users in an encrypted fashion. Now that's on the server side. Let's move on and talk about making HTTPS go fast on the client side.

Making HTTPS Go Fast on Clients

I'm about to load a website, httpversushttps.com, and I want you to watch what happens when I first load this site. Now, this site has loaded over HTTP. We know it's insecure because it's not telling us it is secure, and you're seeing all of these little images load. It's taken about 12.8 seconds. Let's now go and try HTTPS because this is where things are going to get a bit interesting. Wow! That is under two seconds. That is significantly faster. But why is this so? Because HTTPS is meant to be slow, right? I mean, it has to do the negotiation. It has to encrypt on the server, then decrypt on the client. It must have overhead. So why is it so fast? Let's take a closer look and here's what we're going to do. I'm going to hit F12 to open up the devtools. Over to the Network tab, and then let's go back to that HTTP link. All those images are loading quite slowly just behind our devtools there as they did before. And this time, it's loaded in just over 13 seconds, so pretty similar result to before. Now here's what I'm going to do. I'm going to go down and right-click on one of the columns here, and I'm going to enable the Protocol column. This is not on by default in Chrome. Now here's what I want to show you as I scroll back up through these requests. The first is that as we get up to the images, you'll see that the protocol is HTTP/1.1. And as I keep scrolling, you'll see that not only is that protocol the same, but have a look at the waterfall diagram in the right. This is a very traditional waterfall style diagram, where each subsequent image is loading slightly after the one before it. The browser can make multiple connections in some cases, but for the most part, we're seeing these requests cascade down one after the other. And the reason it's taking so long is that we need to wait for all of these images to finish loading. Let's now go and try this with HTTPS, and we'll have a look at those Protocol and Waterfall columns again. Nice and fast this time. It's actually loaded in less than a second. This is not caching either. It makes sure these images are all requested again. Now let's have a look what happens as we scroll down. And the first thing you see is that the protocol is h2. And what we're actually looking at here is HTTPS over HTTP/2. So no longer HTTP/1.1 And when we look at that waterfall diagram, we can see these requests coming down together. Because HTTP/2 allows for a binary stream of content. So we get a lot more data coming down the wire in parallel than what we ever got in HTTP/1.1. Now, I have shared this site and the very favorable HTTP results online in the past and have had people get rather upset at me. It's not fair, they say. Why are you comparing HTTP/2 to HTTP/1.1? This site isn't demonstrating how fast HTTPS is. It's merely demonstrating how fast the newer protocol is. This is not a fair comparison. Except it is and I'll show you why. Over here on Can I Use, we can see how well-supported HTTP/2 is. So about 74% globally. Another 5 1/2% if we include partial support. So we're looking at around about 80% of users globally or if you're down here in Australia, about 93%, and that's a very large portion of the traffic that can communicate over HTTP/2. But let's talk about the support situation for a moment. Because we need to address this issue of speed and talk about why the comparison I just did is a fair comparison. So I'm going to scroll down a little bit, and you'll see that each supporting browser has a little two annotation on it. Let's have a look at what that means in the current version of Chrome. And here it is. This is why HTTPS can be much faster than plain old unencrypted HTTP. Because HTTP/2 is only supported over TLS. Every single browser has got that two annotation, not just for the current generation, but for all the coming versions of these browsers as well. Every single one of them requires TLS. So that example we just saw, that speed test, is a very fair reflection of how fast can you go with HTTP versus how fast can you go with HTTPS. And frankly, I don't really care about being fair. I just care about how fast can I make my site go. And the fastest you can make it go is with HTTPS using HTTP/2. Now of course the server has also got to support HTTP/2, and things can be a little trickier there. So for example, IIS only got HTTP/2 support as of version 10, which launched in late 2016. Any earlier versions of IIS could only run HTTP/1.1. Now there's a way you can cheat this, and I'm going to show you how to do that later on when I talk about Cloudflare. But certainly that is something important to keep in mind. So this is one of these really nice surprising things about HTTPS, not just getting security but getting speed as well. So the next time someone tells you that HTTPS comes with a performance overhead, go and show them httpversushttps.com and set the record straight. Okay, so that's speed on both the server and on the client. Let's now move on and talk about the adoption of HTTPS of downstream services.

HTTPS Adoption of Downstream Services

Think back to the previous module when I showed problems with mixed content. So for example, when I tried to put the IFrame for the YouTube video into the HTTPS page but I loaded it over HTTP, it didn't load and I got a browser warning. And as I explained at the time, this is because if we want to have an HTTPS page, a secure page, we need to ensure that everything that is embedded into it is also served securely. Any one single thing that isn't served securely that's on the page will cause errors or warnings of one kind or another. Now for a long time, this became a barrier to the adoption of HTTPS, and I'll give you some examples of what I mean. Let's start here with Google AdSense. Websites have ads to commercialize their traffic, and for many websites, that is an absolutely essential part of their income stream. Love them or hate them, ads are necessary to keep many publishers afloat. There was a time when Google AdSense could not be loaded over an HTTPS connection. So if a website relied on ads in order to actually pay people their salary and pay for their hosting and run a legitimate business, HTTPS simply wasn't feasible. If they did load the page over HTTPS and then embedded Google AdSense, the browser was going to get very upset and either display warnings or block content outright. It wasn't a tenable situation. But fortunately, this problem is a distant memory. Google's AdSense began supporting HTTPS all the way back in 2013. So it's been running securely for a very long time by now. What this means is that you can embed ads served by Google either over HTTP or HTTPS. So websites can serve ads up over a secure connection without the sorts of problems you get with mixed content. It's not just an issue with AdSense though. For example, the Disqus commenting engine. In fact, I run Disqus on my personal blog. I serve the blog over HTTPS and I embed Disqus in the blog over HTTPS. So their supporting the secure protocol allows me to use them. Now incidentally, my embedding Disqus in my blog caused me a little problem recently. Here's what it was. Whilst I was embedding Disqus over HTTPS, somewhere downstream within the IFrames and the embedded scripts which they put on a site which is using their commenting engine, Disqus was loading a piece of content insecurely, and what that meant was that I was getting the error indicator which you can see here on the screen. That's the one we saw before when we attempted to load active content into the page insecurely. Now clearly me running a blog of the nature that mine is isn't a good look when it has security problems on the page. It wasn't my fault; it was Disqus' fault. They were incorrectly embedding that content, and I had absolutely no control over the way they referenced it. Except I did. Because if we scroll back up, we can see that I fixed it with a CSP, and in fact what I did is I used the content security policy that I showed in the previous module. So I used upgrade insecure request to ensure that even if a downstream dependency that I have no control over whatsoever attempted to load a piece of content insecurely, that content security policy would cause the browser to upgrade the request to the secure version, and it fixed my broken Disqus. So that's actually three great examples right there. Having a dependency on a downstream provider and requiring them to support HTTPS, the fact that this now gives them the ability to impact security indicators in the browser on your site, and then finally that I was able to fix it using a simple CSP in a metatag. So it was a simple very fix in the end. Getting back to downstream services, Pluralsight uses Livefyre for commenting. So very similar story to Disqus. It gets embedded in the page, and because their pages are served over HTTPS, Livefyre has to support the secure scheme, which it does so the page with the comments can load without any browser warnings. One more example is Akamai and indeed any other CDN which is going to load things like images or style sheets or other static material that you want cached, which is then going to be embedded in other pages which may well be served over HTTPS. All the major CDNs will serve their content over a secure connection, and the perceived barrier of downstream dependencies in general not supporting the secure scheme is a perception of a bygone era. We simply don't have this problem anymore with anything major. So that's downstream services done. Let's move on to something else, which still remains a little bit tricky in some cases.

Cost and Complexity

I want to talk about two other aspects of HTTPS that can be a very legitimate barrier to adoption. And after that, we're going to go on and look at two ways of solving these problems. The first barrier is cost. Because traditionally you've had to purchase a certificate. Now over the years, the cost has come down. It used to be an exercise that was much harder to justify with smaller budgets. If you're a bank, it was an easy decision. But particularly if you're a small business or even a personal blogger like myself, justifying the purchase of a certificate wasn't a trivial task. And it wasn't just the initial purchase either. You have to renew the certificate. You have to go and repeat that purchasing process and then do it annually. So every single year, you're paying money for this certificate. Now people would perceive this as a reason not to have HTTPS, and in some cases, that was very valid because even as the cost of certificates came down, particularly for sites that couldn't see a lot of value proposition in serving content securely, the cost became a legitimate barrier. And it's not just the money that had to be paid for certificates. It's also the time investment, and this brings us through to the next point, which is complexity. Think about what's actually involved here. So first of all, you have to actually obtain the certificate. So go out and find a certificate authority, pay them money, go through the effort of doing that. You may well find the certificate is then in the wrong format, so you have to convert it into the appropriate format. That may mean doing things like downloading OpenSSL and figuring out the right commands to get the certificate into a state where you can then configure it on your server, and then that took effort. The certificate had to be loaded. It had to be set up and configured correctly in the target environment. And then as with paying money annually, you have to repeat this entire process annually. Go and obtain a new one, get it in the right format, configure it on the server. All of this required repeat effort. Now let's be perfectly clear about this as well. None of the complexity here is a big thing for good server administrators. None of the costs on the previous slide is any sort of showstopper for organizations of any scale. They're non-events. But of course the internet is made up of millions and millions and millions of sites which occupy realms where both of these actually become an issue. I just mentioned personal sites and small businesses, how to justify the cost. The complexity remained an issue for many small businesses and individuals alike if they didn't have competencies in the right areas. And indeed many services which host web applications, so for example like blogging services, many of those simply had and very often still have no support to load a certificate into. Even today at the time of recording, the Ghost(Pro) service that I use for my blog has no ability for me to load in a TLS certificate and serve my content directly from them over HTTPS. I'm going to talk about how I do that a little bit later on in the module. But first of all, I want to start addressing many of the concerns raised in these two slides by talking about Let's Encrypt.

Using Let’s Encrypt for Free

I just spoke about both cost and complexity being barriers to HTTPS adoption, and per the title of this module, they are now becoming perceived barriers. They're no longer the problems that they once were. Now a large part of the reason why we are overcoming these two problems is what you see in front of you here, Let's Encrypt. And per the title, Let's Encrypt is intended to be free, automated, and open. And it's really those first two attributes that I want to talk more about here. What Let's Encrypt is trying to do is unseat the incumbent CA model. So this is the model where you go to a large certificate authority like Comodo, pay them money, and then manually install their certificate in your environment, and then come back and do the same thing over and over again every year when the certificate is due for renewal. That model created the sorts of barriers we've just been talking about. Let's Encrypt launched publicly around about the start of 2016. And as you can see from the bottom of the screen here, the backers of the project include the likes of Mozilla, Akamai, Cisco, and the EFF. There are many more beyond those as well, but that gives you a good illustration of the types of organizations that are supporting the premise of free and automated certificates. Now the free bit is pretty easy to understand. They simply don't charge you any money for it. But let's drill down more into the automation, and Let's Encrypt provides two main ways of automating both the provisioning and the installation of certificates. And it depends on whether or not you have shell access to the environment that the certificate gets installed on. Let's first of all assume you do and take a look at Certbot. Now Certbot is simply an ACME client. So that is the automatic certificate management environment, and this particular implementation of the protocol is provided by EFF. There are other ACME clients you can use on different platforms as well, but what they all do is run on the target environment and communicate backwards and forwards with Let's Encrypt in order to complete a domain validation process. So in other words, they're making sure that the person requesting the certificate is actually in control of the domain. Now the joy of using a tool like Certbot is that it automates the process, not just the process of acquiring the certificate, but also of installing it and then renewing it. And renewing it is important because all Let's Encrypt certificates only last for three months. But when the renewal process is automatic, it doesn't really matter. It's not like the model of manual certificate purchase where you have to go and put work into it every single time. And in that case, we're normally talking about annual certificates. So once you automate the process, the duration of the cert really doesn't matter too much. We can see that duration on this site, which uses a Let's Encrypt certificate. If I hit F12 to open up the devtools, I'm already on the Security tab, check the certificate, and we can see that the validity period is the 1st of March through to the 30th of May, a nice clean three months. Now an ACME client like Certbot works fine when you have shell access and you can actually run the software directly on the operating system. But what happens when you don't have shell access? What do you do then? Well, that's where you need one of these providers. There's an increasingly large number of hosting providers that support Let's Encrypt as a first-class citizen. So in other words, they've built in the verification and renewal process directly into their platform. And if I jump down a couple of pages, you'll see the sorts of providers that do support Let's Encrypt. And the great thing about these is that you can go to any one of these hosting providers, pay for your hosting, and then easily add Let's Encrypt natively within the environment. It's usually just a couple of very basic steps via an administration interface. Now this is fantastic, and it's a very automated way of getting your certificate. But we have gaps, and those gaps occur when there's no shell access and the hosting provider doesn't natively support Let's Encrypt. A really good example is, at the time of recording, Microsoft Azure doesn't support Let's Encrypt in its app service model. So I can't go and create an Azure website, so that's the platform of the service website, and add a Let's Encrypt certificate. There is an independent third-party site extension that's been built to do it, but it requires a lot of moving parts to all work together, not just to register the certificate, but to hopefully renew it. That's if nothing goes wrong. And at the present time, I would recommend not using Let's Encrypt on an Azure app service. But the thing is for Let's Encrypt, it's still very early days. But be that as it may, they're getting enormous traction and issuing a huge number of certificates. Let me show you just how many. Let's Encrypt publishes stats that are updated daily. And if we scroll down a bit and take a look at the graph, you can see an enormous rate of growth. So again they launched in early 2016, and I'm now recording this in March 2017. Have a look at that growth rate. Only a year ago, they had a mere 2.4 million fully qualified domains that were active. When we scroll forward through to present time, we can see that they've already exceeded 30 million, and that is a massive rate of growth that shows no signs of abating just yet. This is very good news. Because it shows us that more people than ever are getting certificates and going secure. And that's fantastic for all the reasons we've already discussed so far in this course. However, there is also a downside to such easily accessible, freely available certificates, and it's this. HTTPS is fantastic at protecting traffic on the wire. Now that can be great for privacy and great for security, but it can also be great for hiding malicious traffic. So what we're now seeing as a result of Let's Encrypt's success is malicious uses of the service. And you'll see that this was posted back in January 2016, so right at the point where Let's Encrypt first launched. And unfortunately as time has gone by, we've seen more and more malicious use of Let's Encrypt for purposes such as malvertising. Let's Encrypt can also be really effective when running phishing campaigns, and the main reason is that if you can stand up a phishing page where you're asking someone to enter their credentials, enter their bank details or other sensitive information, if you stand that page up and you've got a valid certificate, then unlike this page that we're looking at now, you get a green padlock and a green secure text. It creates confidence in the security of the page. And that's about the only independent indicator that people have to look for. So when they see a page served over HTTPS, they trust it more than a page that isn't. Now that very premise, the one of trust based on the presence of the padlock, is fundamentally flawed. Here's why. This statement is in the story I just showed you, and it's enormously important because it causes us to think differently about what HTTPS means. And the reality of it is that the padlock has got absolutely nothing to do with the trustworthiness of the website it appears on. When a certificate is issued by domain validation alone, it's not telling you anything about the trustworthiness of the entity running the site. Now we'll talk about extended validation certificates in the next module, and they do tell us something very important above and beyond what the statement here does. But for your average user, and think about non-technical people as well, if they see the visual assurance that HTTPS gives them, they're going to trust the site more. So Let's Encrypt is definitely taking us in the right direction. It's a very positive step forward, but it's also not without its challenges, and the big remaining challenge to adoption of legitimate websites is that necessity for the shell access or the hosting provider supporting it. Let's go and take a look at another way of solving the cost and complexity problems but this time by using Cloudflare.

Using Cloudflare for Free

Cloudflare is a different approach again to solving the issues of cost and complexity, and they do a lot more than just issue certificates too. Now I've actually got another course on Pluralsight called Getting Started with Cloudflare's Security, and that goes into a lot of detail about how the model works. So we're just going to spend a little bit of time here in this course recapping on why Cloudflare is a good solution to addressing some of the barriers to HTTPS adoption. Now the first thing to understand about Cloudflare is that they operate as a reverse proxy, and what that means is that if we scroll down the page a bit here, you'll see that they actually run 102 different global footprints around the world, and these are referred to as edge nodes. When you use Cloudflare and they act as a reverse proxy, what your users are actually doing is making a connection to one of the dots on this map. That request is then relayed from that edge node back to the origin server, or in other words, the website where you're actually hosting your application. Now that means Cloudflare can do a number of pretty cool things. So for example, they can operate as a cache. When I look at the map here and I'm down on one of these little purple dots in Australia, if I want to make requests to one of my sites hosted behind Cloudflare, so for example my Have I Been Pwned site which is hosted in the West Cost of the US, I'm making a connection to a very geographically convenient little purple dot. Now very often Cloudflare can serve content from their cache at that very location. So they can send me back things like images, style sheets, other assets that don't need to come from the origin. If the request does actually need to go all the way to the server, so for example, I'm posting some data as part of a registration, then the connection is proxied on to that origin's server. Let's go and have a look at what this means for website traffic. Here's my Cloudflare portal, and I'm presently looking at the configuration for my blog at troyhunt.com. Now if I jump over to the Analytics, I'm going to see some stats on what Cloudflare has been doing with my traffic and what we see here is one week's worth of figures. So over the last week, I had about 1.6 million requests go to my blog. The really interesting thing here is about 1.1 of them was served from cache. So in other words, only about a quarter of my traffic actually had to hit my website. Now that becomes really significant when you look at things like bandwidth. 83 gigabytes worth of data was returned to people browsing my website. Yet only 15 gigabytes actually had to come from my web server. Now that's great because it takes a lot of load off my web server so I can scale further, and it also saves me money if I'm paying for bandwidth. Now that has nothing to do with security, but it's an important introduction to understand that Cloudflare is able to sit in front of the site, intercept requests, and then decide which ones to relay back to the origin website. Now as it relates to HTTPS, because Cloudflare is acting as a reverse proxy and they're really serving traffic from those 102 edge nodes, they can terminate TLS at those points and serve a certificate from there. And if I scroll back up and jump on over to the Crypto section of Cloudflare, we can see this section on the SSL, which of course means TLS. Now what we can see here is that my website currently has an active certificate. So when you go to troyhunt.com, you're actually getting your certificate served up from Cloudflare itself. In fact, if we jump over to my website and then take a look at the certificate that's presently being served, we can see here that it was actually issued to Cloudflare and the CA they're using is Comodo. Now that poses another interesting question. If the HTTPS connection is being terminated at Cloudflare's edge node, what happens after that? So what happens to the traffic that goes from Cloudflare back to my origin server? Let's jump back to the Cloudflare dashboard and have a look at how we can configure SSL. Now I'm not going to run through these here because they're all in the Getting Started with Cloudflare Security course, but we do actually have a lot of configurability here. So for example, you can send the traffic from Cloudflare back to your origin website without any encryption whatsoever. That's one end of the extreme. The other end of the extreme is you can send the traffic from Cloudflare back to your origin web server, which has a valid certificate on it, and all of that traffic gets encrypted along the way. And then there are a couple of middle ground positions in there as well. The important message though is that Cloudflare makes this available for free. I don't pay a cent to run Cloudflare on my blog and serve HTTPS traffic. Now this is really significant because it fills the gap that Let's Encrypt presently has. So that was where we have this situation where you may not have shell access to the web server and your hosting provider of choice may not support Let's Encrypt as a first-class citizen. My blog is hosted on Ghost(Pro), which is an excellent blogging platform, but I definitely don't have shell access. It's simply software as a service so I'm not remoting into any machines, and they don't provide any ability to install your own certificate, either by Let's Encrypt or anyone else. So I have to use a reverse proxy model like Cloudflare in order to serve my traffic securely. Now I'm very happy doing that too, and the analytics show you why. Unlike Let's Encrypt, this is about much, much more than just getting a green padlock in the address bar, and the ability to serve content from cache and serve it quickly to people located close to those 102 edge nodes is enormously valuable. But there's more to this again, and if we scroll down a little bit, one of the things you'll see that you can do with Cloudflare is configure HSTS. Now we just looked at that in the previous module, and that is a really valuable way of protecting your traffic even further. Because Cloudflare does sit in the middle of the connection, they can modify traffic. And in the case of HSTS, what it means is that they can add the HSTS response header as your website loads. So this is great because it's a no code exercise. You don't have to modify your source website or have access to the way it returns headers in order to use HSTS. There's more again. And if we scroll down, we can see things like beta support for TLS 1.3, the next implementation of the TLS protocol. Down further again, automatic HTTPS rewrites, and as it explains here, if you've got references in your application to HTTP that really should be HTTPS, Cloudflare can fix those for you on the fly. Now this is even better than using things like the upgrade insecure request CSP because there's no browser support problems. This is literally fixing the references in the HTML as the page loads. If you've got a website that you want to move to HTTPS, this is probably about the fastest and easiest way to do it. Everything from the fact that this is just a five minute setup process to get behind Cloudflare through to the fact that it's free, although as you can see by the blue button here, there are certainly higher plans you can access and pay for, and then of course, all of the other security features that we haven't delved into here and the ability to do things like caching and performance optimizations. But there's one more thing I want to show you about Cloudflare that ties back to one of the earlier points in this module. Let me show you what that is. I'm here on my most recent blog post, and this particular post has got a bunch of images on it. Now because I'm using Cloudflare, any request to troyhunt.com is going to go direct to them. Because this page has a lot of static content on it, they can serve a lot of that directly from their edge nodes very quickly from cache. And because they can do that, there's one other really neat thing they can do that relates to both HTTPS and performance. I'm going to press F12 to open up the devtools, and then I'm going to give this page a hard refresh to ensure it requests all of the content from my site via Cloudflare again. Let's now scroll up to the top of these requests and have a look at the Protocol column. There's a heap of content coming down over h2, so HTTP/2, which of course we know you can only use over HTTPS. So earlier on when I was talking about the performance benefits of HTTPS by virtue of the newer HTTP protocol, I mentioned that you need to have server support for it. Well, when Cloudflare serves content, they support HTTP/2. So it doesn't matter whether your origin website supports the newer version of the protocol or not. When you're pulling content like the images and style sheet that we see loaded here, I'm getting the benefit of that faster protocol. And we can see by looking at that waterfall chart just how much more efficient it's made this page. That is a lot of content loading in parallel. I love using Cloudflare because it solves so many problems at once. It's not just about getting a free certificate. It's about all the other positive security and performance aspects you can get from using a reverse proxy that's able to have this fine-grained control over the traffic. There is no better way to solve the cost and complexity problems than by putting a site like this behind Cloudflare. With that said, let's move on and summarize the module.

Summary

Let's summarize the module. And the most important takeaway from all this is that those traditional barriers to HTTPS simply aren't what they once were. There was a time when there were many legitimate reasons that HTTPS was a difficult thing to use, lack of support on dependent services, for example, the cost of certificates, the complexity of managing it all. We've got great ways of addressing all of those today, and almost all the arguments against using HTTPS are simply no longer an issue. It goes further than that as well. Because we saw how HTTPS can actually be a virtue well beyond just the security benefit it provides, and of course I'm talking about the support for HTTP/2, making your traffic go fast. That's a really positive thing to have. And yes, it's HTTP/2 that makes it go fast, but you can't get HTTP/2 without having HTTPS. And no, I don't care that it's not fair. I just want to get fast. I spoke quite a bit about Let's Encrypt, and they are doing fantastic things. They have really turned around the CA industry. They've made encryption accessible to so many people who had difficulty accessing it before. Both good people and bad people, and that's always the nature of technology in general. But they are fantastic, growing very quickly, and well worth it if you want to run certificates directly on your site. But I do love the Cloudflare approach. I love it not just because it's easier than ever to get certificates on your site regardless of shell access or support by your hosting provider, but I love the fact that they can do really cool things with the traffic, serve it from cache to unburden my server and at the same time make it fast for people spread around the world via those 102 edge nodes, and then make it easier for me to do HTTPS right. So make it easy to turn HSTS on or automatically write responses to have the proper scheme referenced in links in the page. That it's a five minute setup process that's available for free makes it a really compelling argument for securing sites. So that's overcoming what we now well and truly know are the perceived barriers to HTTPS. These things are simply no longer a problem. Let's now move on and do one more module. Let's go beyond the basics.

Beyond the Basics

Overview

It's time for the last module of the course, and this time, I want to go beyond the basics and talk about a number of aspects of HTTPS that we haven't already covered. So particularly things that go beyond the realm of what you would normally do as a developer. But I also want to go and cover a few other aspects of HTTPS we haven't really touched on yet, and wrap up by sharing a couple of pieces of good news, good progress within the industry. Let's go and take a look at the overview of the module. As I just mentioned, this course has been very developer-centric. That's the whole point, It is What Every Developer Must Know About HTTPS, but I also want to help you as a developer see some of the other aspects of HTTPS which you wouldn't normally be exposed to, but are actually quite useful things to know. So the first thing we're going to do in this module after the overview is look at a bunch of acronyms that we haven't already touched on. I also want to talk about some of the other logistical and social aspects of HTTPS, so some of the things we can do to make it stronger, and also some of the human elements of HTTPS. So particularly, how it works with people's psychology and indeed the good sides and the bad sides to that. Because there are some risks we need to discuss as well. Now just risks, but a number of remaining blind spots in the way we use HTTPS. So aspects of it that are easy to neglect and indeed, some ongoing weaknesses, even when we get pretty much everything else right. And finally, I want to come back and talk about progress, and as I just mentioned, even whilst I've been recording this course, some things have changed. Some very positive things have changed. And normally, I don't like to see things change whilst I'm recording a course, but in this case, they illustrate many of the things I've been saying absolutely perfectly, and I'm very happy to share those with you here at the end of this module. So let's jump into it and start talking about some TLS and HTTPS acronyms.

TLS and HTTPS Acronyms

As with so many areas of technology, TLS and HTTPS is full of acronyms. Indeed, they're acronyms themselves. But there are many other terms we haven't even touched on in this course that are worth well being aware of. So I want to go through the nine that you see in front of you, very briefly. I'm just going to define each one and give you one or two sentences on why they're important. And we're going to start with SNI. Now SNI is Server Name Indication, and the primary thing that SNI allows us to do is have multiple certs on the one IP address. Traditionally, you would need a different IP address for every single cert, and that started to make life pretty tricky infrastructure-wise if you had lots of sites and certs that you wanted to consolidate your hosting. Next up is SAN, which is Subject Alternative Name and what we're really talking about here is the ability to have multiple domain names on the one cert. This is what Cloudflare does, and if you go to troyhunt.com, for example, you'll see that there are other subject alternative names on the same certificate. I'm going to give you an example of that a little bit later on in the module. And clearly this offers an upside in terms of consolidation of certificates. Next is PFS, which is Perfect Forward Secrecy, and what we're doing here is ensuring that even if a private key for a certificate is compromised, past sessions whose traffic was encrypted with that key, don't themselves become compromised. So PFS is an important way of protecting traffic in the event of an incident such as a key compromise. DNSSEC, the Domain Name System Security Extensions. There's been a lot of talk about DNSSEC for a long time, and what we're talking about here is protecting against the forgery of DNS records. Now clearly, if you can forge a DNS record, then you can route traffic to other locations, so DNSSEC is an important part of the integrity of the domain name system. Next up is DANE, the DNS-based Authentication of Named Entities, and this is where, at the DNS level, you can specify the certificate keys. Now this helps us tackle things like the compromise of ACA, and I mentioned DigiNotar back earlier on in the course. Using DANE also requires DNSSEC, because that helps ensure the integrity of the DNS records themselves, and this is one of several security defenses we're seeing at the DNS level. Another one of those is CAA, the Certificate Authority Authorization, and this is where we're using DNS to whitelist the CAs that are allowed to sign certificates for the domain, so once again, to protect against a CA compromise like with DigiNotar. CCRL, the Certificate Revocation List. Now this is maintained by the CA, and it lists all the certificates which they've had to revoke. Now this is really important, because if a certificate is compromised, then the CA needs to be able to instruct clients to no longer trust it. Not dissimilar to CRL is OCSP, the Online Certificate Status Protocol. Now this is actually an alternative to CRL, and it allows for real-time and near-real-time revocation checks. An important tangent to add here is that OCSP stapling has become very important, because it solves the overhead of clients talking to the CA directly when they do their revocation checks. And instead, it becomes the site itself which does the revocation check and returns the status within the TLS handshake. Bit of a tangent we don't need to go off on, but useful information all the same. And lastly, PKP, or Public Key Pinning. And this where a client has a predefined list of public keys that it can accept. Now this is the public key of the certificate, and this is a good place to move on and start talking about HPKP, or HTTP Public Key Pinning. Let's go and delve into that, and that will allow us to talk about the value proposition of this public key pinning.

Public Key Pinning

When your browser loads a website over HTTPS, and it gets a certificate and it validates that the certificate is legitimate, just because the certificate checks out and your client trusts the CA and does the validation, it doesn't mean that it's actually the correct certificate, and what I mean by that is that it may not be the actual certificate that your site is serving up. What if there was a man in the middle that had compromised a certificate authority, DigiNotar style, and served up an incorrect certificate for that site? It's a valid certificate, it checks out, it's from a trusted CA. But what if it was fraudulently issued? Let's talk about how HPKP tackles that. What we're doing with HPKP public key pinning is defining the public keys that are permissible for a domain. Now I'm going to show you precisely how this works in a moment, but let's go and talk about some of the attributes of HPKP. So, for example, there is a max age, and what we're going to do here is say this is how long the client must enforce these keys for. The max age ensures that if within this period of time a different certificate is served up for a domain, the client won't trust it. HPKP also allows us to define whether it applies to subdomains as well. So is it just for the parent domain or if you have other sites beneath that, running on subdomains, does this policy apply to them as well? And finally, HPKP facilitates reporting of violations, and what this means is that if a man in the middle manages to insert a certificate which doesn't have the correct public key, one of the ones that has been pinned by the client, there is a construct within the spec that allows that violation to be reported. So that's the overview. Let's go ahead and have a look at how it actually works. HPKP is implemented as a response header, and it's simply a header that's called public-key-pins. Now the most important attribute of this response header is the hash of the public keys. Now you can have multiple hashes of multiple public keys. It can be the public key of the certificate itself, the public key of an intermediate certificate, the public key of the certificate signing request. Multiple different ways you can implement these pins, and indeed, you can have multiple pins themselves. You may have backup certificates. You may pin both the public key of the certificate and the one of the CSR. I'm not going to go into too much detail here, because my Pluralsight course on browser security headers has an entire module on this, so go and check that out if you want to more about HPKP. Let's just run through the remaining attributes here. So I mentioned max age before. This is four weeks'-worth of seconds. You express the max age attribute in seconds. And what this means is that if the browser has received this public-key-pins header before and it's got a whitelist of pins, and then it sees another cert, one whose public key is not pinned in this whitelist anywhere within the next four weeks, it's going to reject it. Then there's the include subdomains attribute, and just like the attribute of the same name which we saw when did HSTS earlier on, this does exactly what it sounds like it does. Now also, back when we looked at HSTS, I talked about the TOFU problem, the Trust On First Use problem, and with HSTS, we could solve this by pre-loading. We have the same problem with HPKP, but we also can't preload it, so there's a little bit of a gap here in terms of HPKP and getting one good request that returns those valid pins such that they can protect subsequent requests. Last attribute here is the report-uri, and what the report-uri does is it takes a path where the client should submit any violations to. So if the client has previously received this header and it's pinned the collection of public keys, and then it sees an incorrect public key, one that's not within the whitelist, the client will automatically submit a report to the report-uri explaining exactly what happened. Now again, I go into a lot more detail on this in the Browser Security Headers course, including the situation with browser compatibility. So go and check out that course for more details. So that's public key pinning and the implementation over HTTP. You can still do public key pinning without using HPKP. So, for example, Chrome pins a number of Google certificates and that is actually baked in to the browser. But for the most part, if you're working with websites and browsers, this is the implementation that you'll be using. Let's move on to something different. Let's go and take a look at SSL Labs.

SSL Labs

This is SSL Labs, and it is here to help you test your TLS. We've had this discussion about the interchangeability of the terms SSL and TLS, and indeed, HTTPS. So you know what this really means when you see it say SSL. Now the great thing about SSL Labs is it makes it really, really simple to test your HTTPS implementation. So all we do is enter a host name. So, for example, my blog. Now what's going to happen is this service, provided by Qualys, is going to go and look at how my website is willing to talk over HTTPS. Now this is not doing any sort of malicious tests on my site. It's simply looking at how the site has been configured to communicate securely. This normally take a few minutes to run, so let's let it complete, and then we'll come back and take a look. So the test has finished and Qualys has found multiple IP addresses behind this domain name. Now as we can see in the right-hand column, they've all been graded A-plus, so good news on all. Let's just go and drill down on this first IPv6 address and have a look at what's been found in the report. Now the most obvious thing on the screen here is a big A-plus rating, which you would rightly assume is a very good result. The reason I want to show you this report though is it helps you see just how much goes into an HTTPS implementation. Now before we even scroll down the screen here, we can see a couple of things towards the bottom, which we've already touched on. So, for example, this side works only in browsers with SNI support. We defined SNI just a little bit earlier on, Server Name Indication. And if someone was using a really old combination of browser and operating system, so for example, Internet Explorer 6 on Windows XP, they wouldn't be able to load my site, because that didn't support SNI. We can also see down the very bottom that this site implements HSTS. We've touched on that, we know what it is. That is part of why it has such a good rating. Let's scroll down a little bit. You'll see here that this certificate supports alternative names, so this is a SAN certificate. I mentioned that just before, and we can see that there are other host names on this cert. So this is what Cloudflare does, it combines a bunch of them together. Now these certs include other sites of my own, so for example, cloudflareonazure.com. That's from my Cloudflare Security course on Pluralsight. My wife's blog, I can see kyliehunt.com in there, and worldsgreatestazuredemo.com, which we've already looked at. None of the other ones on that list of alternative names are mine, but it doesn't matter, there's no security risk in having these other names on there with the way Cloudflare has implemented their SAN certs. A little bit further down, we can see extended validation, no. This is not an extended validation certificate, but we will talk about what they are or whether or not they matter right after we finish looking at this report. Down a bit further we can see revocation information about CRL and OCSP, we just defined those, as well as whether I've implemented CAA on this cert. And again, we just defined that and no, I haven't used CAA. That's not presently supported by my DNS provider. Down further and we can see the supported protocols. This is one of the things I talked about very early on in the course, SSL Versions 2 and 3, and how they then led into the TLS versions. And I mentioned back then that we should not be using any SSL, and here we are with the SSL Labs website doing the SSL test, making it very clear that I am not using SSL anywhere, only TLS. Have a look also at the cipher suites beneath those protocols. I spoke about cipher suites when I was talking about the negotiation phase of the client and the server, communicating backwards and forwards. So here's everything that my server is presently supporting over TLS 1.2, and as you can see here, in server-preferred order, here's all the 1.1 cipher suites, and the 1.0 suites as well. Further down again and we're seeing whether or not the site is vulnerable to attacks such as DROWN, BEAST, POODLE, and a POODLE attack could come over SSLv3 or TLS, neither of which is a problem on my site, but you can use SSL Labs to see if it's a problem on yours. I've got forward secrecy enabled, I've got OCSP stapling enabled, and as we saw earlier on, I've also got HSTS enabled but I don't have HPKP. And I talk about some of the reasons why not in that Browser Security Headers course. Now all of this looks like a lot of information, and it is. It's very useful information, and it's the sort of information that your system administrator would normally take care of. Because I'm using Cloudflare, they've taken care of all of this. The only thing that I've actually configured is HSTS and I configured that within the Cloudflare portal, so this was a no-code configuration on my part. So thinking back to the previous module on overcoming the perceived barriers of HTTPS, and when I spoke about the effort of configuration, you can see how much it is to potentially have to set up and also why I was quite enthusiastic about letting Cloudflare do it for you. So that's SSL Labs. Let's now move on and start talking about extended validation certificates.

Extended Validation Certificates

We spent a bit of time on this site earlier on as we secured the application. And it is now set up securely, you can go to World's Greatest Azure Demo and you get a green padlock, a green secure text, and of course, the green-scheme HTTPS. This is now a well-configured site and we get confidentiality, integrity, and authenticity. But think about that authenticity component for a moment. All we really know is that the site, which is at worldsgreatestazuredemo.com is the one that is actually loaded into the browser here. We know absolutely nothing about who runs worldsgreatestazuredemo.com. That is not what the padlock tells us. And this is one of the problems with the perceptions people have about that little visual indicator. It gives us no assurance whatsoever about this actually being a legitimate, honest, trustworthy website. Let me give you an example for what I mean by that. Let's talk about the trouble with padlocks, and I want to give you an example. Imagine this. Now this domain is currently available. There's nothing to stop you from going out and registering it. There's then nothing to stop you from going and getting a free certificate from Let's Encrypt or from putting it behind Cloudflare for free. And if you did that, and you stood up, say, a PayPal phishing site behind it, this is what people who went to that site would see. So they would be there at your malicious phishing site, literally seeing secure in green with the green padlock and the green HTTPS all up there in their browser. They would have a much higher degree of confidence in the legitimacy of the site by virtue of these visual indicators. And indeed, this was the point I made earlier on about Let's Encrypt, where we're starting to see malicious uses of certificates. You see, the problem is that with both Let's Encrypt and Cloudflare, and most HTTPS implementations you see up there, you're merely doing through a DV process, a domain validation process, where they're making sure that somebody actually does own the domain before they get a certificate for it. Compare that to the legitimate PayPal site. Now this looks quite similar, but clearly there is a difference. We're actually seeing PayPal, Inc. next to the padlock and then the jurisdiction that they're operating in, in this case, in the US. This is an extended validation certificate, also known as an EV cert, and to get this, PayPal has had to go through a much more rigorous process, and they've had to actually prove their identity before they could get the certificate. Now what this means is that when you go to the legitimate PayPal site, you get this visual indicator that makes it perfectly clear who is actually running the site. Now there are a couple of things we've got to be clear about here. Firstly, the certificate on the bottom is no more secure in terms of the cryptography than the certificate on the top. You can take a domain validation site and an extended validation site, go and plug them both into SSL Labs, and you'll see that they come out identical, assuming they support the same cipher suites and protocols and everything else. The point is, the one on the bottom is no more secure than the one on the top, so what do you actually get? Well, here's how the CAs explain it. This is the way the GlobalSign website positions an extended validation certificate. They say, "Hey, if you go and get one of our EV certs, "your customers are going to have more trust in your site, because it has your name on it and it proves who runs it." The thing is though, imagine this. Let's bring that malicious PayPal one back in. When you think about your average, everyday person, so people who use the internet day-in/day-out, but may not be technical people, are they really going to be any less likely to trust the site on the bottom than they are the site on the top? You see, one of the things that's been changing is that the way browsers are representing DV sites is getting closer and closer to the way they represent EV sites. GlobalSign's example at the top is actually an older version of Chrome. When we look at the newer version beneath, you're actually getting a lot more green than what you used to. There are many people now saying that the value proposition of an EV cert is much less than what it once was, and a lot of that is because visually, they're getting closer and closer, and your average, everyday person doesn't understand that it really just gives you confidentiality, integrity, authenticity and doesn't actually mean that the site isn't malicious. They don't get that. So whilst there's clearly a value proposition in people seeing the green padlock and the secure text, there may not actually be a whole lot of value in showing them an EV cert. Now of course, that's a point that the CAs may not like to agree with, because they would like to sell EV certs, and arguably, there is some benefit to having the name of the organization up there. So by all means, go and get an EV cert. They're a lot cheaper than what they used to be too, although they certainly are more expensive than a DV cert. But in this day and age, it doesn't pose quite the value proposition that it once did. So that's extended validation certificates. Let's go and have a look at a few other reasons why you really want to have HTTPS.

Other Reasons to Adopt HTTPS

I want to touch on three points quickly here about other value propositions of HTTPS, and one of those is search engine optimization. Now this is a very ancillary benefit that's not really directly related to security, but a little while back, Google did this. This was back in August 2014 and Google made the call that the presence of HTTPS merited a slight bump in your SEO. It's not a big bump, and in this particular article, they do say that it certainly doesn't carry the same weight as signals such as high-quality content, but it's a bump all the same, and considering the lengths that some people will go to to improve their search engine optimization, something as positive as HTTPS also giving you an increase in your searchability is a pretty good outcome. Let's look at another reason to adopt HTTPS. Now this comes down to the way browsers pass the Referrer header when you follow a link from a secure site to a non-secure site, and it all comes down to the implementation of this RFC. Clients should not include a Referrer header field in a non-secure HTTP request if the referring page was transferred with a secure protocol. And what they're really trying to do here is protect potentially sensitive information which may be in the URI of that secure page. Now you really should try and avoid doing this anyway, but imagine you had a secret in the URI of your secure site, and then somewhere on the page, you had a link to a non-secure site. Any man in the middle monitoring that traffic would then be able to observe the secret if it was passed in the Referrer header. So this RFC means that that won't happen, and not only will it avoid sensitive data being passed over a non-secure request, it will avoid any information being passed over a non-secure request, so the site that you linked out to over HTTP won't see where your users have come from. Now that use-case works fine, however, if you link from an HTTPS website to another HTTPS website, and there's no longer a man-in-the-middle concern, then the Referrer header is sent. So it doesn't complete absolve you of the risk of sending sensitive data to another party. Now there is actually a Referrer policy security header which you can use to define how and when Referrers are set, so that's what you really want to be looking at. But this is just an interesting aspect of how HTTPS impacts that particular header when people click links on the site. Let's look at one more upside of HTTPS. You can only use Brotli compression if you're serving the content over HTTPS. Now Brotli is a new compression algorithm from Google. Let's go and take a look. Google came out with Brotli in 2015, and as I just mentioned, it is a compression algorithm and it happens to be an exceptionally good one. In fact, as you can see here, Google claims that this format gets you 20 to 26% higher compression ratios than what we've seen in the past. Now this is very good news. Reducing your traffic by about a quarter can make a pretty significant difference, not just to website speed, but also to things like your costs if you're paying for bandwidth. But as per the slide just before, you can only get Broti compression with HTTPS. Now you've still got to enable Brotli on your server and then it's not supported by all clients, but again, it's one other thing that you can only do with HTTPS, and as with the bump in SEO, it's really not at all related to the security advantages of implementing TLS on your website, but it's nice to have these other reasons which drive us forward, which push us towards going secure, even for reasons that aren't security related. So, that's a few other good reasons to go HTTPS. Let's have a look at a few areas that people often miss as they move their sites to HTTPS.

HTTPS Blind Spots

Here's a really good example of what I would refer to as an HTTPS blind spot. You may be familiar with Ashley Madison, not because it's a site that you may frequent, but because they did make quite a lot of news in 2015. This was one of the most well-publicized, most significant security incidents we've seen to date, and it all came down to Ashley Madison having a really serious data breach. Now the point I want to make here is that Ashley Madison serves everything over HTTPS. They used HTTPS extensively in 2015 as well, but that didn't stop them losing tens of millions of customer records. Now as a technology person watching this course, you understand that the presence of HTTPS on a site like this has absolutely no bearing on how secure the backend is, but that's not the case for your average person, and I've touched on this several times in the course and indeed just in this module. When someone sees HTTPS in the address bar, and in this case they see an extended validation certificate as well, that increases their confidence level in the security of the site. Now some of that is due, because of course there are security advantages to HTTPS. We just spent a whole course talking about it. But there's the risk of creating a false sense of security because as we well know, it's not going to stop data breaches from happening. Let's look at another little blind spot. We spent a bit of time on this site earlier on in the course, securing it properly, so we fixed things like mixed content. But there's another little problem here, another blind spot and it's not affecting any of our visual indicators on the site. But have a look at this. There's a link here to my website, and it's going to an HTTP address. So if someone was to click on this link on my website, a man in the middle could potentially intercept this request. Now my website does use HSTS, but it doesn't preload it, and the reason it doesn't preload it is because I've got other subdomains on troyhunt.com which I don't want to serve under HTTPS just yet. So what that means is we've got a risk here, where someone could click this link, make an insecure request, and be man-in-the-middled. So the blind spot here is that I really should be going through this site and upgrading this particular link to use the HTTPS scheme. Now this is the sort of thing that easily slips through the cracks, because functionally, it all still works. There's no visual indicators in the browser to suggest I've got a problem. It's just one of those little edge cases which you should get right as you move into HTTPS. Let's have a look at one more blind spot, and I want to go and check a bank. This is Barclays in the UK, and as you can see from the address bar, it has not been loaded over a secure connection. Now if you want to log in, and I hover the mouse over the log-in link, we can see that the URL down in the bottom left-hand corner of the browser will take us to a secure page, but the thing is, we can't trust this page, and in a way, it's a little bit like the conundrum of loading a log-in form over an insecure connection. Yes, it may link us off to somewhere secure, but you first have to trust the page which loaded it in the first place. So what if a man in the middle saw the initial connection to barclays.co.uk and decided to change that log-in link to go to their own malicious website. Now it is, in fairness, one degree removed from actually asking for log-in credentials on an insecure page, but you can see the problem, and for an entity the likes of a bank handling financial information, this is the sort of thing that you'd like to get right, and it brings you to the point where you have to start saying, "How many upstream resources of the thing that I really need to protect also needs protection?" The landing page of a bank is really the sort of thing that could be served securely. Because customers are going to use this page to then go through and access very sensitive things. So this is a big of a blind spot, and it's the sort of thing that particular banks, but really, any website that wants to do HTTPS well needs to consider. Now you may remember earlier on in the course, I showed a different bank doing the same thing. I want to go on and talk about one more thing in this module, and show you what has changed with that bank just whilst I was recording this course.

The Tide Is Rapidly Turning

This is the exact image that I showed back in the first module, and I talked about the exact same problem that I just covered with Barclays. So that is, loading a bank over an insecure connection. But let's go and have a look at HSBC now, because there's something a little bit different. You're now seeing the HSBC bank in the UK load over a secure connection, and this is very good news. But have a look at what we see up in the address bar now. We don't see the padlock. All we've got is the information icon and HTTPS grayed out. Clearly, something is wrong. Let's press F12 to jump into the dev tools, and if we scroll up a little bit, we can see the problem here, and it's this image. We know this issue, we've seen it before in the module on securing the application. HSBC has elected to embed this one image insecurely and it's taken away all of the green bits from the address bar. So we know this problem very well. We've looked at it in detail, and I wanted to include it here right at the end of the course because it shows two things. One is that just over the single-digit period of weeks that I've been recording this course, websites have moved forward very quickly. So that's good news. We're seeing HSBC go secure. But two is that, as I've said many times throughout the course, there are lots of little nuances in terms of how HTTPS is implemented. Now HSBC could have easily mitigated this problem with something like an upgrade insecure request content security policy. That would have fixed this and it speaks volumes to how many little issues you can have when moving to HTTPS. Let me show you one last thing to illustrate how quickly the industry's moving forward, and it's this. I'm very happy to now see my national airline serve their website over an HTTPS connection by default, and if I do now choose to go and log-in from the home page, I can do so with confidence that it hasn't been manipulated at all in transit. That is great news. And it was only a few short weeks ago that they were demonstrating the behavior that I showed you at the beginning of the course. So as I mentioned in the overview, this is one case where I am very happy to be able to say that during the process of the course, something changed, because it changed for the better. And as with HSBC, although differently to HSBC, Qantas doesn't have mixed content warnings, but regardless, both of them have shown just how fast the tide is turning on HTTPS, and that is a very positive note to finish on. Let's go and summarize the module.

Summary

Let's summarize the module and indeed the course. And of course, I started off talking about a whole bunch of different acronyms that we won't go through again now, but I hope that was somewhat useful. One of the things that we covered in this module that I do rely on quite frequently is SSL Labs, and it's really, really handy for getting a very quick snapshot of the HTTPS implementation of a website. So definitely use that it if you're transitioning though to HTTPS or even if you just want to have a look at what the state of your exiting implementation is. I spent a bit of time talking about extended validation certs, and I used the word trust quite a bit. And obviously those certificate authorities selling EV certs really like to talk about trust. But I also explained some of the social aspects of this, and particularly how normal domain validation certificates are building trust in a much greater, much more visible fashion than what they did in the past. So EV certs may not actually pose a lot of value over DV certs in this day and age. I spoke about blind spots as well. So the little things that can be easy to miss in an HTTPS implementation. So the consumers of the service, assuming the presence of the padlock means the site is secure. We understand the nuances between securing the transport layer and securing the things behind it, but it's worthwhile remembering how your users perceive it. Linking out to other websites over HTTP. Fix those, upgrade those outbound requests, and serve them securely. And of course, incidents like Barclays which we just saw where the landing page is insecure. You really want to get all that upstream content as secure as possible. But lastly, as we saw with HSBC, even though they had a little issue, but particularly as we saw with Qantas, the tide has turned. We are seeing a rush of sites going HTTPS and that's where we started the course, talking about the likes of The New York Times, and that's a great place to finish it as well, because now is the time to go to HTTPS. There has never been a better time. And I hope this course has been able to help you understand how to do that efficiently and effectively using the modern technologies we have available to us today. Thank you very much for watching. I'm Troy Hunt.