# #174 - OWASP Top 10 Web Application Attacks

[00:00:00]

[00:00:00] **G Mark Hardy:** Hello, and welcome to another episode of CISO Tradecraft, the podcast that provides you with the information, knowledge, and wisdom to be a more effective cybersecurity leader. My name is G. Mark Hardy. I'm your host for today, and we're going to be talking about the OWASP Top 10 Web Application Security Risks, and what you can do about them.

This is an important topic because if you're not in the stone age, you probably run a website, or at least your organization has, and it could be things such as marketing pages to promote your brand, could be e commerce sites that allow customers to buy things from your company, could even be a discussion board where you interact with customers, could even be a customer relationship management or CRM system.

It doesn't matter what kind of website you run. What matters most is the website that you run needs to be secure. Nobody wants to have their customer [00:01:00] information breached. Nobody wants to have their products emptied out of their warehouse at one cent each. So given that overarching goal, let's look at how attackers think and act to understand our websites.

I'm going to first introduce you to the Open Web Application Security Project. Also known as OWASP. Now in their own words, the OWASP Foundation works to improve the security of software through its community led open source software projects, hundreds of chapters worldwide, tens of thousands of members, and by hosting local and global conferences.

It, wasn't always that big. It was launched back in 2001, became a U. S. non profit in 2004, and their website talks about their vision, no more insecure software, And their mission, to be the global open community that powers secure software through education, tools, and collaboration. And today, OWASP is the world's largest non profit organization that is concerned with software security.[00:02:00]

Now, before researching this episode, I was mostly only familiar with their top 10. They have over a dozen flagship projects, including 91 different cheat sheets, a full stack bill of materials, a web testing framework, just to name a

few. Check it out. Very own OWASP Threat and Safeguard Matrix, the TASM is there as well.

So over lunch someday, go to OWASP. org, browse around their extensive resources. You might find something that you could really use. Now, web app vulnerabilities exist. But why? Why are they there in the first place? If we go back and we look at things For example, programming languages. PHP might have come out in the early 90s, had a lot of vulnerabilities, but over time versions were released that didn't allow you to write vulnerabilities.

And as a language matured, you tend to get better. Now in Java and Python, they don't have the same guardrails. Definitely if you're not, if you're in C. And as a result, what you find then is that over time we have a [00:03:00] significant growth in website requirements. New companies want them, companies want more sophisticated ones, so we add new programmers.

Now, over the years, programmers get more experienced. But every year we add new web developers, and therefore, if we could do a plot of the vulnerabilities written to years of experience as a web developer, I think we would see that as an inverse relationship. But with this geometric increase in websites over the last 20 years, and these languages that are being used that have the ability to, well, program bad things into them, we're going to be behind the power curve, and it's going to get worse.

Because now we have AI generated code, but also AI generated code remediation. GitHub Copilot will look at your code, look at the public repos, identify similar vulnerabilities that might have been patched in the past, and say, hey, here's a code fix that someone else did, just do a merge commit, and then you don't have to write your own code.

Wow! That really lowers the bar in terms of the sophistication of your developer. But also makes you wonder, could there be a [00:04:00] malicious poisoning of GitHub to cause bad updates? Can you go ahead and deliberately put a whole bunch of similar common vulnerabilities, put a patch out there that actually kind of doesn't fix it, but makes it worse?

Let this AI tool train on it, and then before long, your vulnerabilities are becoming everybody else's vulnerabilities. And from a legal perspective, you can't hold the AI too accountable. I think Air Canada tried that recently, slamming, oh yeah, there are. ChatBot made up an answer and we don't want to stand behind it.

And rather than just tell the customer, look, we're a gazillion dollar company. You'd like to fly with us. We'll, sorry, we'll make up a few hundred bucks because you had a bereavement issue. They went and they fought it in court and then lost at which first of all, seems that that was kind of a bad choice from a customer relations, but secondly, it does drive home the point that.

People are what are accounted in legal systems, not software, not hardware. I've never seen a case of a computer being sentenced to having half their memory chips removed, and then their clock speed stepped down for six months as a punishment [00:05:00] for something happening. Now, with the number of vulnerabilities going up nearly exponentially because of the ability to discover them and the number of programmers that are writing them, let's try to get inside the minds of how an attacker would look for one of these things.

and take advantage of it. Let's say you want to find a web app vulnerability. You're going to do a few things. First of all, you're going to download a web application security testing tool. It could be Burp Suite, the OWASP ZAP, the Zed Attacks Proxy. There's a couple that come to mind and can fuzz your website.

What does fuzzing mean? It means that you can go ahead and send a lot of different values of a particular parameter, hoping that one of them will work. And the fuzzing then allows These tool sets to try different combinations much faster. And sometimes in ways that people can't do or wouldn't think of in doing it.

Let's say, for example, that you have attackers. They're trying to bypass your client side security or think of it another way. Let's say we'll make it simple. We'll create a [00:06:00] clever menu as a dropdown box. You can only click on number one to 12 for the month of the year. Most users only have the ability to click the dropdown box and pick the number.

Once the user does that, they hit submit. It's going to post. or upload that value to the server. Now what a tool like Burp Suite or Zap is going to do is it's going to act as a proxy between what the browser sends and what the server receives. Now, if you wanted to change that value from 1 to 12 to something else like 0 you could you could put any value in you want because you bypass the client security controls Because instead of the client just sending it to the server you put burp in the middle You know, stop free hit open it up alter the things around a little bit Package it back up again and send it on its way.

And so as a result, what the browser sent is not what the server receives. Never trust user input, or more precisely, I should say, always validate user input. You can trust it, but you need to validate it. Trust, but verify. [00:07:00] And in addition to changing values, you could do Ability like brute forcing multiple values.

Let's say your developers pass values in the URLs. A URL, a simple one, might be yourwebsite. com sent invoices sent customer 123. Well, I could use a tool to go ahead and iterate through customers 000, 001, 002, etc. to 999 using a technique called forced browsing. And so in addition to all these other features, these tools have extensions which allow software developers to Upload all kinds of other features into the tool to increase its ability to, well, find web application vulnerabilities.

Hopefully, that's a high level understanding of what a web fuzzing software like a Burp Suite or a ZAP can do. Now, let's take a look at some of the common vulnerabilities for which researchers and criminals are on the hunt. The most respected list are of web application security risks come from OWASP.

When we introduced a few minutes ago, it's commonly known as the OWASP Top 10. It doesn't get updated every year. There's a 2010 version, A 2013 [00:08:00] skip to 2017. Then 2021. I remember a few years ago I was at a security conference and ran into the person who was managing that. I said, how come you don't update these things every year?

And he said, nothing changes. Well, it does change, but it doesn't change that fast. It's not like you get a basketball pool or all of a sudden is it a whole different bracket than it was different from last week. They tend to be fairly consistent, although they will change places with each other and they'll move around a little bit, but the list represents a broad consensus about the most critical security risks to web applications.

Now the most recent version, which is three years old and they seem to be on a three to four year refresh cycle, so we'll see if this is still current, but this is for 2021. Broken access control, number one. Cryptographic failures, number two. Injection, number three. Insecure design, number four. Security misconfiguration, number five.

Vulnerable and outdated components, number six. Identification and [00:09:00] authentication failures, number seven. Software and data integrity failures, number eight. Security Logging and Monitoring Values, Number 9, and Server Side Request Forgery, or SSRF, Number 10. Now, I'm not expecting you to

memorize that. But, It is a quick overview, and I'm going to dive into each one of those in a little bit more detail, so you get your head around it.

So, although you may not be able to regenerate that list, when somebody mentions any one of them, you'll know what you're talking about, or at least you know how to listen to it. Number 1 was Broken Access Control. Now access control is what allows someone the permission to visit a site and the pages or the components within a site and we need it because we'll authorize paying customers who can come in and then get access what they should to.

Non paying customers may see less and criminals we should basically keep out of our sites. Now, just because we let a paying customer in doesn't mean, well, that they need to see everything. They need to see the things that we want to sell them. For example, we ran the Amazon Web Store. We want to show the items that they want to buy so that we can make a sale.

And additionally, we want them to be able to log in and [00:10:00] see things that are unique to them, like their purchase history, account information, password information, etc. Now, this private information should only be visible to the customer and internal to the company staff for the customer help desk.

Unfortunately, this is the most common web vulnerability found to date. Developers forget to put in access control checks everywhere on their website. So if we go to the earlier example where attacker notices a URL naming convention used by a website appears follow a specific pattern. They might use a forced browsing technique to enumerate other locations that would allow them access where they shouldn't have.

And this can get pretty bad when PII data is exposed from customers or an attacker can force browse into an admin page that allows them new capabilities for which they should not have any access. It can also lead to exposure files on the server if the attacker tries to use directory traversal. Having the URL slant dot dot or dot dot slant dot dot dot and so on the URL it could to potentially let them see the password files [00:11:00] Get dot dot dot dot dot and then come back down again etc password or database connector files It'll show the username and password to log in or people will sometimes go ahead and put comments in code that could be traversed Viewed, and those comments have, oh, here's the ID and the password for a test site or a production site.

And it's not theoretical, I have actually seen that happen before. As a result, we want to take a look at the type of recommendations that OWASP has. to remediate these issues. The recommendation is to implement a deny by default

approach. The answer is no, unless we specifically say yes, but no one has access to anything unless you're on an approved list.

Now OWASP will also recommend using things like the cross origin resource sharing or CORS control on your site. CORS defines a way for client web applications that are loaded in one domain to interact with resources in a different domain. So here you create an allow list of what websites can request data [00:12:00] from your site.

The most important control to overcome broken access control is to use stateful session identifiers. Session identifier is given almost like a token to say, hey, this is how you can go ahead and continue to interact with this website because we've given you an identifier. Stateful says we want to keep track of that every time.

I don't want to issue it once and then it becomes like a bearer instrument that if I could replay it or put it on a different machine, it would work. You want to know someone's identity on every single login. And then you take this identifier as a way to broker every data call to say, Hey, Is this person on the approved access list to visit this website or this resource?

Sound a little bit like zero trust for the web, isn't it? But if not, you can return a message that says not authorized or send them someplace else. Or if they're not a paid customer and you want them to pay, said, Hey, you could sign up and here's, here's what the rate is. Now it's also good to log these events so you can see who's poking around your website in a non normal way, as well as where these events are happening.

Intrusions are coming from, [00:13:00] and, if they're coming from inside your organization, we would want to take a look at that as well. Now, another important control you can use is rate limiting. Humans can't click a key a hundred times a second, but software like Burp Suite or Zap can spider a site and crawl through hundreds of these fuzzing scenarios in a very short period of time.

Now, if you limit the API and the controller access. Then you can minimize the harm from these tools. Think of it this way. If an attacker is used to running a spider on a website that takes about five minutes to do thousands of queries, but your website limits the API calls such that the spider has to wait several seconds, each query times thousands of queries That's going to prevent them from finding more parts of your website to exploit in any reasonable amount of time.

If it takes two weeks to fuzz your website versus five minutes, the attacker is probably going to move on to a softer target, and that's also a good win for your organization. Your regular users, okay, it takes two seconds instead of two [00:14:00] thousandths of a second. Most people stick around for two seconds, but again, we're trying to go ahead and throw a little bit of sand in the gears for the attackers.

Let's look at the second web application security risk. It would be cryptographic failures. I mean, let's face it, security has gotten better over the years, our understanding of math has gotten better over the years, and computing power has gotten better over the years. If you're using older encryption ciphers with short key links, They're potentially subject to brute force attacks today, and they should be deprecated, meaning you don't use them anymore.

Smart security researchers found there were inherent flaws in a lot of protocols. Early versions of SSL, Secure Sockets Layer, TLS 1. 0 had some problems, 1. 1 had some problems, Transport Layer security. And if that wasn't enough, there are users that use protocols that were never even designed to send things encrypted, like HTTP and SMTP or FTP.

Or if you're a fan of Hacker Jeopardy, Telnet, Port 23. These things send text in clear text. And so as a result of these cryptographic failures, attackers who can sit in the middle of the [00:15:00] communications channel can find ways to see, potentially even alter, sensitive data. Even worse, what happens is they could include malicious content.

That then gets brought into your system or off to the user because it is trusted because they said, well, the other side wouldn't do anything bad. So here's one such example of this. For a long time, many Mac app developers used the Sparkle framework to update their applications. And prior to version 1.13 in 2016, Sparkle used only HTTP traffic. Now, the attackers saw that your Mac was running applications like Adium app and Evernote note taking app and Handbrake Video Transcoder or Slack. All those need to be update, they might choose to perform a man in the middle attack. When your fully patched Mac says, I want the new update of Evernote or Slack, the attacker says, well, here it is.

Here's a new update. Just open this new DMG Apple disk image file, which happens to be malicious. And it's a very clever way to bypass a fully patched Mac because the [00:16:00] user is already prepped to install software and keep their machine updated. And they were given a software update and in it went. Now that we better understand this.

Encryption threat. What should we be doing? First, don't ever try to roll your own encryption. My saying, and I'm a mathematician, I've done a lot of crypto math in my background. In fact, when I started my own business, my very first client had rolled their own encryption. And they said, we want 80 hours of your time to validate that no one's going to break it.

I did, and then that was within the first 40 and they said, okay, well, kid, here's another algorithm and you don't get more time. And I broke that too. And probably not understanding a lot about how to run a business. In my twenties, I went ahead and I gave them proof of concept code written in basic. Okay.

Bad idea because you, first of all, embarrass your client in front of his boss and his boss's boss. And although I recommended using a well established crypto algorithm at the time, DES data encryption standard was the, Algorithm I had suggested, [00:17:00] I never got any follow on business. And I was kind of wondering until a little bit later when I got older, I said, yeah, that was kind of bad form.

But the point was, is that some people out there can look at these things and nobody has an ugly baby. If you roll your own crypto, it could go bad. So don't let your people invent their own encryption. Most of these stories end quite badly. Use a modern cipher. If you use Transport Layer Security TLS version 1.2 or version 1.3 Once you do that, run a free tool like the SSL Server Test Tool from Qualys to inspect your configuration. It'll give you a rating and provide recommendations on what you can do to improve your encryption keys and protocols. Because if you find yourself using a bad protocol or one without security, Encryption, please upgrade it.

If you have instances of FTP, File Transfer Protocol, use SFTP or FTPS, they're different. And there's no defensible reason to use an insecure protocol when an equivalent secure protocol is available. If you find yourself in the unfortunate instance where you can't get off [00:18:00] of a bad protocol, it's embedded, you can't change it, try to wrap This unencrypted protocol message with encryption.

For example, let's say it was a vendor product that only supports Telnet for logging in. Might wrap your Telnet traffic inside of the open source proxy service S Tunnel, which will add TLS encryption to that session and therefore Telnet effectively gains encryption. It's not the optimal approach, but it's a lot better than remotely accessing a resource over the internet and providing your username and password in clear text.

I know you could also wrap things in a private VPN to achieve similar results, but at least you're thinking about the right stuff. Our third web application security risk is injection. Let's just be number one a few years ago. Injection flaws occur when an application sends untrusted data to an interpreter.

The most common example of this is a text field on a website to which a user adds input. For example, you might have a field where a user enters your ID, enter your password. And usually what happens on the back end [00:19:00] is the website takes the username and password fields, performs a database query to match these values to an existing entry, and then allow a login.

And database queries can be manipulated. See, a query typically is going to return a true or false. If this thing, then do that. All right. An example here, the database query might trigger the following action. The query returns true, then log in the user. If the query returns false, then say no, or ask the user to log in again, or lock them out, whatever you want to do.

But triggering a database query that always returns true could be a good way to bypass all these login security controls. So let's use an example, a basic login page that's written by an inexperienced developer. Because remember, your software building materials is not going to list that Bobby the intern wrote this code over the summer and it got sucked up into GitHub into the production.

Let's use an example of a basic login page that Input name and password value from, then use the SQL statement. So I'm going to, because mostly this [00:20:00] is an audio, don't worry about like, we don't look at the video on the statement, but something like select star from table of users where user ID is quote, name, quote, and the password is quote, pass, quote, semicolon, because you're going to collect name, you collect password, you're going to fit in there.

And then when you get a match, you're going to go ahead and grab the right entry from the table. Okay. Well, if the user enters in their name, enters in their password, it's correct. Good. If one or the other is wrong, it fails. It doesn't work. Now, what if the attacker, instead of entering their name, puts in, let's say, Ross, single quote, or one equals one semicolon hyphen hyphen, is that the username?

Now this SQL injection attack, as it's called, results in the following command being executed. Instead of selecting star from users where user ID is quote name quote, and password is quote pass quote, select star from users where user ID is quote Ross, and then in the text field it was quote or one equals one, semicolon, hyphen, hyphen, quote, and then password [00:21:00] equals pass.

Well, the unfiltered data input now becomes executable code, and the OR condition is always true. And the semicolon hyphen hyphen makes the balance of the valid original SQL statement to become part of a comment. So it's ignored when it's executed, and the bottom line is the statement is always going to return a true value. Even though you didn't provide a valid password. Oops. Now this can get worse because there are other scenarios rather than just login. Maybe an attacker can log in as an admin and change the pricing of a shopping cart item from 499 to 0. 01. Or if you're not going ahead and checking the validation of the user input, someone says, Hey, here's your shopping cart.

I go ahead and I order, reorganize it, take the burp suite, change the price. And then when I push it back up, the server says, Oh, I put it there. I gave it to the user so they can look at it. They gave me a credit card number. It must be right. No, we need to go back and revalidate that. So let's prevent that from happening.

If we know that a bad actor is going to be fuzzing text fields, what should developers be doing about it? And the best approach, there's several different things, but parameterizing and [00:22:00] validating user inputs. And do this always on the server side, because on the client side, you can play with things. theoretically speaking, I used to have a one character domain name.

I probably should have held on to that. I could have retired on that, but that was back in the 90s and they didn't charge for domains back then. And it was client side input. You can only have, had to have three or more for that particular TLD. And I said, what happens if I change the three to a one? And I entered in zero as the value and it took it and I got the zero domain.

And then the next year they started charging and I was like, yeah, I don't want to pay for it. That's what happens when you think one year ahead instead of 25, 30 years ahead. Nonetheless, the whole idea was, is it was client side. And no matter what that server thought, it. Believe that whatever it gave me as a guardrails would stay in the guardrails.

So now we could do and take prepared statements in, let's say, Java. Instead of having code that just says get the username and password from the form, inquire the database, add an extra step. We'll create a variable that's equal to the [00:23:00] SELECT statement from your database. It might look something like select the account balance from user data where the username is question mark and then create a prepared statement that uses that variable and returns the results of it so that nobody's injecting things into that particular query code.

Now we have a link to an example in our show notes. It's on CheatSheetSeries.owasp.org. There's a whole bunch of them out there. I mean, there's about 90 different sheets that you can look for. But essentially, what this is doing is it's splitting the answer from the form into different strings. It's going to void out anything that might be added after the first variable.

And without getting too technical, using prepared statements should be the default. after taking input from every single user query. By the way, for anybody who likes Randall Munroe and XKCD, go ahead and look up xkcd. com slant 327, exploits of a mom, is a nice way to illustrate SQL injection.

Our fourth web app security risk is [00:24:00] insecure design. This is related to design and architecture flaws. Let's take a look at a couple examples to get a better understanding of what we're talking about. Let's say we're going to host an online website such as Ticketmaster, where customers can buy tickets to a sporting event or a concert.

Attackers might try to DDoS your website, or use bots to buy lots and lots of tickets they could then scalp. If you find yourself in this scenario, then you need to have something like this. like an AWS web app firewall with bot control to minimize the common attacks that you can expect. Now, as we look into this risk, we can see it's actually a lot broader than you might think.

For example, you might have a website that collects customer information. Amazon store is a user section that users can go to look at their previous orders and track delivery. It's a good thing that provides value to the customer. And they have search functions. What were my previous orders? Do I want to buy this again?

And how do we design that search function so it doesn't search for things that might be private or private to other customers or things that are in Amazon? So knowing the idea of how to do that allows us to go ahead and avoid the risk of [00:25:00] insecure design. I could talk for hours about all the different, Implementations have been secured to 9.

Just leave it there to make sure that as you're designing things, you put security in at the beginning. Don't try to paint it in at the end. It's not cost effective. Our fifth web application security risk is security misconfiguration. Now in order to do security right, we usually need to do three things.

One, never use unpatched software. Up to date. Number two, ensure custom code doesn't have vulnerabilities. Bobby the intern writing the code, sorry. And

then finally, avoid misconfigurations. That's what we're going to talk about now. Because there's a lot of opportunities to misconfigure. You can misconfigure your operating systems, your Java frameworks, your middleware, your load balances, your web app settings, etc.

I mean, here's an example. Security Assertion Markup Language, or SAML, is an open standard for exchanging authorization and authentication information. It's a good standard overall because it allows you to standardize your logins across multiple websites. As with anything, though, it can be [00:26:00] misconfigured.

You see, SAML is supposed to work like this. User tries to access a resource. The service provider sends the user to an identity provider with a SAML request. ServiceProvider forwards a SAML request to IdentityProvider, who authenticates the user, sends a SAML response back to the user, which is forwarded on to the ServiceProvider, and the user is authenticated.

Basically, you're getting a ticket that says, hey, this person is valid. But if SAML is misconfigured, things can go wrong very quickly. For example, an attacker might be able to fool the ServiceProvider by using unsigned certificates or self signed certificates, if they're not being checked. You can use an XML signature wrapping attack to adversely modify the message structure.

So, the application logic and signature validation module are going to be parsing different parts of the message, allowing the XML signature to verify successfully, but causing the application logic to process the bogus element. Another misconfigure. An animation that could take place with SAML is responses to XML documents.

Attackers can use tools to [00:27:00] generate XML external entity attacks, or XMEs, for example. There's a BURP suite extension called SAML Raider that could be used to trick the server to send the ETC shadow file instead of the XML document. Now, this Misconfiguration now is allowing credential stealing from the target.

Since we've seen multiple ways that something can be misconfigured, where should we look for the proper ways to find out how to configure something? I mean, I would recommend a couple places. One is the Center for Internet Security or CIS Benchmarks, cisecurity.org. You can go there and find recommendations for most IT systems, such as Windows, Linux, Oracle, etc.

Even download hardened images from CIS CIS Benchmarks already meet the benchmarks, which will save you the time it takes to configure them. Now use those as your golden images for every Windows server or Ubuntu Linux server and watch your systems become a lot more secure. Defense Department had something similar.

they called DISA a Secure Technical Implementation [00:28:00] Guidance, or STIGs. Now when I was in the military, we had the STIGs and you'd find documentation that was similar to CIS on things how to configure Active Directory or Windows Group Policy Objects and things such as that. If they are available to you, and I'm not sure what the distribution limits are today, then it might be valuable.

I mean, if it's good enough for DoD, it's probably more than good enough for your business. Finally, don't forget to ask the vendor, the manufacturer. Usually their websites have a configuration guide for administrators with all the configs, all the settings. Find that and follow it. and implement it. You might even ask the manufacturer to validate that your implementation is secure.

And this is highly recommended if you have something like Salesforce that has a lot of things can be misconfigured. Oh my goodness. I am astounded by the number of settings that are in Salesforce. I, I am not a master of it. I can poke around as an admin. I keep something running, but there are just too many details in there that are beyond my level of training and expertise.

I'll admit that, but nonetheless, at least you should know what you don't know. [00:29:00] These types of capabilities or requirements have given rise to a whole suite of tools like AppOmni, AdaptoShield, Obsidian Security, that look at your third party SaaS tools for misconfigurations. The buzz term is CSPM or Cloud Security Posture Management, if you want to investigate platforms like that.

Further, full disclosure, I have been a customer of Obsidian Security for a number of years and I really like it. It works well. Number six, moving on, our web application security risk is vulnerable and outdated software components. Consider this, software ages like milk, not like wine. It doesn't get better over time.

Just remember, you have to discover all the areas that are vulnerable. And the problem is you generally need multiple tools to do it. Now, for most organizations. They'll use a vulnerability management tool like Nessus, Qualys,

or Rapid7, there's whole family tools out there that'll scan their system for vulnerable or outdated software.

But you need to understand how these tools really work behind the scenes. I mentioned this a couple episodes ago. They'll have an agent [00:30:00] that runs commands. If they're on a Red Hat machine, they might run a command like Yum lists to show all the installed packages. And they just simply look up those packages against the information contained in the CVE details list to say which packages have a known vulnerability findings.

The problem with that is if a developer doesn't use Yum to install a package, but just use an App Get and install or whatever, then The vulnerability agent might miss it. And if there's a firewall rule of preventing the agent from talking to the server that aggregates all of the agents and their data, you don't know that the machine was vulnerable because that never got to you in the report.

What's worse, you might also have things like application libraries and Java that are hidden to the vulnerability management tool. You could be running bad versions of Apache struts or even Open SSL. So here's some guidance. Create an SBOM, a Software Bill of Materials, to know what you have. For example, if you're programming in Java, try using the Versions plugin for Maven.

This helpful plugin allows you to display All of your dependency, know what libraries you're using. What's even better about it is you can find other [00:31:00] libraries that you're using, which are not up to date. I didn't say vulnerable because you see a piece of installed software might be on version one, the manufacturer might now be on version five and anything older than version three is no longer supported.

Try that with your old Windows 7 machines, right? And it's not just operating systems either. So even though there is an unknown CVE, you probably want to be on a version of software that is supported. Let's say versions one through four had a bug in the product that eventually got caught and got patched in version five.

However, it's that bugs might not have received a CVE, but if you run version one, two, three, or four, you're still at risk of encountering these bugs. Updates come and go, but vulnerabilities accumulate and therefore use the latest stable version from the manufacturer to best ensure high availability for your system.

Another important thing to do to help your organization against stale software is to leverage automated testing software. Development team knows that they need

to update stale [00:32:00] software. The next question you want to answer is, does this update break my system? Having the ability to run a quick script which can answer that question in minutes, Avoids analysts spending hours on manual testing that might eventually show the same break.

Having a real time inventory of your software is more important than knowing what's in GitHub. We highly recommend having a vulnerability management tool. And a Runtime Application Self Protection or RASP tool to identify vulnerabilities in production, not just over on the dev side. Moving along, number seven, the seventh web app security risk is identification and authentication failures.

Note that our previous SAML example of security misconfiguration risk, could also fall under this category. We have some more examples to get you thinking about what could possibly go wrong that involves this risk. The first is credential stuffing attacks. If there's anything we've learned from data breaches, it's that many users choose to use insecure passwords and they use them over and over again in a lot of places.[00:33:00]

Now bad actors know that as well. So they'll either download a list of compromised passwords, from the dark web, and try them with its same user information in other places, or they'll try to guess common passwords and hope they work. If you're in March Madness season and you happen to know that somebody is a fan of a particular school, you might want to go ahead and figure that some variation of that might be in the guess password for people, depending upon what your complexity requirements are.

But does a attacker write a bot or a script that's going to automate guessing in the hopes that you can get in? One tool for that is THC Hydra. This tool is part of Kali Linux and therefore it's commonly used against passwords. Another example is how bad actors can bypass conditional access policies.

Some developers fail to write good conditional access policies for Microsoft authentication. Now, if you're not familiar with conditional access policies for AD, now called Entra ID. It's basically an if then else statement that looks [00:34:00] for things like user location, device information, applications they're using, or real time risk judgments to say what happens next.

And this is known as a grant control. It says, For example, make a device pass MFA, or make the device be required to follow Microsoft's Intune configurations, or it has to have the latest update, or it has to be on the list of

items that are not out of compliance. Otherwise, require password change, require something else, don't let them in, etc.

Now, one of the configurations that's commonly used for is device platforms. Conditional access policy that says only apply this to, say, Windows, Mac, iPhone, or Android devices. I have a client where everything is either Windows box Or it's an iPhone. There are no Macs. There are no androids. So somebody trying to log in for a Mac or an Android, hey, stop them right at the door.

They don't even have to get past that first check because we know that they're not using an approved device. Now, that doesn't work all the time. Why? Because it's based off of a [00:35:00] user agent string which could be modified by an attacker. Remember our Burp Suite when we were doing a web content? Well, now we can use tools like MFASweep, which is a Microsoft PowerShell script. It's going to try to log into various Microsoft services and APIs up to 10 different times when you're provided with a username and a password, hopefully a valid one. That's really what it's for. But what if I try to log in as a Windows user?

What if I log in as an Apple user? What if I log in as this, I get things like that, et cetera. And now if the MFASweep finds that you thought you had MFA turned on everywhere, But a particular interface such as the Microsoft Graph API doesn't require an MFA challenge and the attacker could potentially use an emulator mode to change the user agent string from the web portal to the graph and bypass the MFA requirement.

Now, they still got to go ahead and get in with the user ID and a password, but remember, we're already at the point where we think the attacker knows the password. And now all they had to do was get around MFA. This gets them around it. Another way this attack is occurring is via a tool called road recon to gather information about your [00:36:00] active directory and then use the road recon plugins for policies to see an HTML file that shows how AD is configured.

And once attackers know what the policies allow, they can look for loopholes on how to come in and bypass your identification and authentication. The big thing here, perform threat modeling and test to confirm you're as secure as you believe you are. You might start with the road recon tool to see what your policies are and how these could go wrong or be misconfigured.

And start with a single policy such as the one that blocks All non U. S. logins. Then ask the team, can someone please try logging in from a VPN with a

outbound node overseas to confirm. Did it block the login? Okay, perfect. Now let's test the next policy. This one by one testing these different abuse cases is a great way to know if you're as secure as you think you are.

Our eighth web application security risk is software and data integrity failures. Attackers who cannot log in. get into our companies directly, we'll try indirect means, including software supply chain attacks. It might attack a [00:37:00] server upstream. One example is the attack that took place on CodeCove.

CodeCove provides analysis of pull request comments so that a developer can quickly check their pull request coverage and risk, and we don't even have to leave the development workflow. What are we looking at? But attackers broke into that server and captured the environment variables that were being uploaded from customers CICD environments.

Now, long story short, it is Kind of led eventually to, for example, HashiCorp losing their private GPG key. Oops. Another type of supply chain attack could be theft of credentials. Let's say your password manager gets hacked, and its software hosts something malicious that steals your private credentials.

You log into your password manager next time, it'll capture your inputs in cleartext, send it to the attacker. That would be a big problem if you don't have MFA from a separate provider. or something that's hardware based like a YubiKey. Another example is an attacker does something like typo squatting or domain squatting, meaning what?

They find a URL that just expired and then register it themselves. Or find something that looks just like the domain that you're trying to target, but maybe a letter is off or a [00:38:00] zero instead of an O. And then some unsuspecting developer searches and finds that, or chooses a library from the attacker thinking, Hey, this is legit stuff and it's not the real one and bam, attacker malware gets installed and gets incorporated into the code and out it goes.

And we want to ensure that we verify and validate who's sending us data. And I did a whole episode on supply chain attacks on episode number 66, entitled Working on the Supply Chain Gang. So if you want to learn more about how to stop these attacks, be sure to check that out. And also a couple of weeks ago, we did have an episode with

Cassie Crosley. on software supply chain security. After the episode, she was kind enough to send me the book. It's available on O'Reilly, and that episode is just a couple back. So plenty of resources there. Our ninth web application

security risk is security logging and monitoring. Let's say a criminal organization hacked your server that sits on a box with PII data.

Do you have to declare a data breach? Well, it depends. See if the data has been exfiltrated outside of the company or not. [00:39:00] You have an incident, but you need to look at the logs to see if data were stolen. Now, if you don't have any logs, chances are your legal team is going to say you need to disclose and you got to deal with all the legal consequences that come with it.

But if you do have logs, you say the attacker went here and they went here, but then CrowdStrike caught them and stopped them and they were not able to exfiltrate data. That's a great story to tell a leadership team. It's a great way to go ahead and put a boundary on what you think the breach might have been.

And therefore you can say, Hey, we looked at the logs and they got in, but they didn't get there. So make sure your logs give the insight you need. And if you think it's going to take 30 days to perform an investigation following a ransomware incident lasts for 21 days, then you should have logs that are like 60 days in duration or longer.

Don't go cheap, buy only 14 day logs, only to find out the bad actors realize that they waited 15 days after planning their stuff before launching their attack. And now you have no idea how they got in. Every backup version you have has got that malware in it, and you're probably going to be paying a ransom.

Another common risk you need to mitigate here is logs that are [00:40:00] stored only locally. If the attackers come in and wipe out the logs, you can't tell what happened. It's all gone. The evidence is gone. So make sure your logs are stored in a separate place, and even places where your developers don't have access to, in case you have a developer that goes rogue.

Our recommendation for secure logging is the following. Perform a tabletop exercise. We do tabletop exercise. You want them, we'll come help you out with it. It says bad actors in your crown jewel database. How do we know what logs could show the actions that have occurred? What should we do next time we think some of the data has been tampered with?

And we do all those scenarios. Now, the 10th and final web application security risk is SSRF. Server side request forgery attacks. This is the attack that got Capital One. Essentially, a bad actor was able to find that a server running the mod security web app firewall was misconfigured. And the tool which is supposed to protect the server had a forward proxy instead of a reverse proxy.

Well, forward proxies look okay on security scans, so it wasn't flagged. And this allowed the attacker, however, to query the AWS EC2 [00:41:00] metadata service, and then steal security credentials. This is a server side request forgery. And by using this attack, the attacker could then use the AWS command line interface to log in with those credentials and start stealing private data from Cap 1's S3 private buckets.

If you'd like to simulate this attack, there's a tool from Rhino Security Labs called Cloud Goat, where you EC2 SSRF to get hands on experience. If we know this is an issue, what should we do about it? First, standardize. It's a lot easier for the security team to analyze 10 configurations than 1, 000 different ones so you can identify vulnerabilities such as SSRF.

Reuse proven patterns that avoid introducing new risks so that if you do find a problem, you can roll it out and fix it everywhere fairly quickly. Next, look at a defensive depth approach. For example, use AWS Access Advisor to reduce the permissions to which things have access. Amazon has a tool called GuardDuty.

They will act as a data loss prevention tool and raise an alarm [00:42:00] when abnormal amounts of data are being exfilled. Use a web app firewall that can detect and prevent server side request forgery attacks. You never know which of these might be the one that saves your bacon during an event. So don't just rely on one alone.

All right, let's recap. We looked at the 10 most common, doesn't mean it's the limited 10. There's 11, 12, 13, but this is the Pareto principle. This will get the majority of problems on there. Broken access control, cryptographic failures, injection, insecure design, security misconfiguration, vulnerable and outdated components, identification and authentication failures, software and data integrity failures, security logging and monitoring failures, and server side request forgery.

Those 10 now should mean something to you. Not that you could explain them in detail, but when someone mentions an SSRF, Oh, okay, fine. Yeah. Somebody's going ahead and grabbing this credit. Got it. All right. Misconfiguration. It means something. Good. You're now better versed in this. And I hope you've enjoyed learning about the [00:43:00] OWASP Top 10 Web Application Security Risk.

Like anything else in our career, it's hard to protect things we don't understand. So if you have a staff or a team or people who want to learn more about security, you want them to learn more about security because they work for you,

pass along this episode. And have them go ahead and check out CISO Tradecraft.

And thanks for listening. We now are over 27, 000 followers on our LinkedIn page, and we had almost a quarter million impressions in the past month, and we couldn't be happier. We just want to say thank you for listening to our content and watching our content and helping us get the message out about how you and other people can improve their cybersecurity journey in their career.

If you want more and you're not doing it already, please come to our LinkedIn page, CISO Tradecraft. and subscribe. We put good information out on an almost daily basis in the business thing, but it's a very high signal to low noise. We don't put out junk. I don't tell you what we had for breakfast, but it's things that you might want to be aware of that can help build up your knowledge and your resource.[00:44:00]

So again, thank you very much for being a part of our CISO Tradecraft audience. And as always, feel free to contact us either at cisotradecraft. com or through our LinkedIn page. If you've got some feedback or things that you'd like to learn, or you want to engage us and have us help you out with some projects.

My name is G Mark Hardy. I'm your host for today. Thank you very much for listening or watching. And until next time, stay safe out there.