O'REILLY®

Early Release

RAW & UNEDITED

Identity & Data Security for Web Development

BEST PRACTICES

Jonathan LeBlanc &
Tim Messerschmidt

Identity and Data Security for Web Development

Best Practices

Jonathan LeBlanc and Tim Messerschmidt

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Identity and Data Security for Web Development
by Jonathan LeBlanc and Tim Messerschmidt

Copyright © 2016 Jonathan LeBlanc, Tim Messerschmidt. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-04-01: First Early Release
2016-04-08: Second Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491936948* for release details.

978-1-491-93694-8

[FILL IN]

Table of Contents

Preface

"Companies Lose $400 Billion to Hackers Each Year"[1]

—Inc. Magazine

In a cybersecurity market report[2], issued by Cybersecurity Ventures in Q4 of 2015, they stated that cyber attacks are costing businesses between $400 - $500 billion a year. In the same thread, IT security spending is due to increase by 4.7% in 2015 to $75.4 billion USD, with an estimate that the world will spend upwards of $101 billion in information security in 2018, and growing to $170 billion in 2020. Due to this, it's projected that there will be a cybersecurity workforce shortage of 1.5 million by 2019, as demand is expected to rise 6 million people in 2019.

As web and application developers, designers, engineers, and creators, we are no longer living in an age where we can offload the knowledge of identity and data security to someone else. In this age, a web developer can unwittingly open up a security flaw on a site simply by not understanding how to properly obscure data in transmission, a project manager can cause a major attack vector to open up in an application by not understanding that previously secure password algorithms have been shown to now include flaws, and not prioritize the work on rehashing the database of user records. It is now the business of every person working on a system to take a part in ensuring that your users and data are protected.

Despite these facts, it seems like every week we have new cases of companies, from startups to massive corporations, losing privileged user information, credit card data, medical records, and many other pieces of information that are entrusted to protect. It comes to light that many of these same organizations never took the effort to encrypt data properly, storing everything in plaintext, awaiting for some hacker to abuse.

1 *http://www.inc.com/will-yakowicz/cyberattacks-cost-companies-400-billion-each-year.html*

2 *http://cybersecurityventures.com/cybersecurity-market-report/*

The true problem is that hacking is no longer just the business of individuals wanting to prove that they can breach a system, it is now a realm of organized businesses, hacking for money or to damage the business.

This is where this text comes in. As we explore each chapter and concept, we will be taking the approach of how to plug holes in existing systems, protect against viable attack vectors, and how to work in environments that are sometimes naturally insecure. We'll look at concepts such as:

- Understanding the state of web and application security, with concepts.
- Building security password encryption, and how to combat password attack vectors.
- Creating digital fingerprints to identify users, through browser, device, and paired device detection.
- Building secure data transmission systems through OAuth and OpenID Connect.
- Using alternate methods of identification for a second factor of authentication.
- How to harden your web applications against attack.
- Creating a secure data transmission system using SSL/TLS, Synchronous and Asynchronous cryptography.

In the end, we'll have a comprehensive understanding of the current state of identity and data security, knowing how to protect ourselves against potential attacks, and protecting our users from having the data that they entrusted to us compromised.

Acknowledgments

First of all we would like to thank the O'Reilly crew for publishing this book and enabling us to share our knowledge, thoughts and opinion with many individuals around the world. A huge special thanks goes out to our editor Meg Foley, who has been patient, supportive and helpful throughout the process of finishing this work.

Our thanks also go out to both Lenny Markus, Allen Tom, and Aaron Parecki, who patiently reviewed this book's manuscript and helped to improve its quality tremendously.

We'd also like to thank our developer relations team for proof-reading, providing critique and freeing us up to work on this book.

Finally, we'd like to express our gratitude to you, our readers, for buying this book. We hope you enjoy it!

Jonathan

I'd like to start out by thanking my partner in crime, Tim, for being an amazing co-author to work with. Without our continued conversations, building up and breaking down all of our ideas into new amazing hybrids of their original selves, this book wouldn't have been what it is today. Your ideas, drive, and humor made this one of my favorite experiences.

To my wife Heather, you've helped to keep me sane when I decided to write my first book almost 5 years ago. Despite the fact that I forgot how much time away that took, you stood by me when I decided to write another one. Without you, I could not have kept my sanity and drive throughout this process. You have always been by my side to encourage me to chase my dreams. You've been my biggest advocate through all of this, and I love you for that.

To my daughter Scarlett, throughout the time that I have had to pleasure to be your father, you have brought a calming effect into my life. With constant chaos, you have allowed me to see that the world doesn't have to be as serious as I used to think it was. You've brought a peace into my life that I will always thank you for.

To my group, my friends. We may all go our separate ways, be split through companies and across the world, but I will always see you as some of my closest friends. We have been through so much together, and have sacrificed a lot. Despite all that, you have been our supporters through everything we have gone through, boosting us up, allowing us to succeed. Thank you.

Tim

I'd like to thank Jonathan, who's not only been a fantastic colleague and friend, but also a great co-author on this book. It was brilliant to be able to bounce ideas and thoughts back and forth and I am positive that the book would have been far less interesting without your influence, support and work.

My wife, Karin, deserves a huge thank you - and probably an even bigger bouquet of flowers - for granting me all the time I needed in order to finish my work on this book.

Joe Nash, Alan Wong, Steven Cooper and Cristiano Betta have been a fantastic team throughout the time of authoring this book and deserve to be mentioned here.

I am grateful for everyone that encouraged me to write this piece and saw me rambling about security concepts and usability concepts on various stages.

A special mention goes to Danese Cooper, PayPal's Head of Open Source, who highly animated me to write down my thoughts beyond blog posts.

Finally, I would like to thank both John Lunn and Taylor Nguyen, who supported me tremendously in writing this book and gave me support and advice throughout my career.

CHAPTER 1
Introduction

Jonathan LeBlanc and Tim Messerschmidt

One of the most important investments that you can make in a system, company, or application, is in your security and identity infrastructures. We can't go a week without hearing about another user customer breach, stolen credit cards, or identity theft. Even though you can put an entire series of hurdles in the way of a potential attacker, there will always exist a possibility that your databases will be breached, information will be stolen, and an attacker will begin attempting to crack the sensitive data that was stored (if encrypted).

There is no bulletproof, secure method for protecting your data, identity and data security has always been about mitigating risk, protecting the secure data, and buying yourself enough time to take action and reduce damage if something like this should ever happen to you.

As we dive down into the concepts, technology, and programming methodologies behind building a secure interface for data and identity. We're explore the decisions, tradeoffs, and core concepts that you will need to understand as you embark on making those final decisions about your security.

As we begin, the best place to start is in exploring what the major problems with data security and identity are in the industry right now.

The problems with current security models

The current state of industry security is not one in which the technology can't keep up with the potential attack vectors, it's one in which development choices lead us down a path of weak systems. One of the biggest mistakes that many of us tend to make is to assume that a user will understand how to protect their own accounts, such as with strong password choices or two-factor authentication, or even if they do, that they wouldn't pick the most usable choice over the easiest one. We, as developers, have to protect our users in the same way that we try to protect our systems, and we must assume that the user will not do that for themselves.

What this means is that we have to purge a few misconceptions from our heads:

- **The user will always use the most secure options**: The simple fact is that the worst thing you can rely on is that the user will be capable, or willing, to use the option that will secure them and their data. The onus has to be on the site or service owner to ensure that data provided by the user for their security (such as a password) is hardened to ensure that minimum levels of security are imposed (see more about data encryption and security in Chapter 2). For instance, when two-factor authentication services are offered, a typical adoption rate is approximately between 5-10% of users.
- **We should always make systems more secure, at the cost of usability**: This is typically one of the reactions to the last point, to make a system as secure as possible, at the cost of usability of the system for the user. This is simply not the case, there are numerous mechanisms that can be put in place to enhance security without drastically affecting the user. We'll explore this further below.
- **Our security will never be breached**: From startups to large companies, many engineers have put too much faith in the security of their systems. What this has lead to is lax data encryption standards, meaning that personal and privileged information, such as credit card data, home addresses, etc, are stored as cleartext, data that is not encrypted in any way. When the system is breached, hackers have to put in no effort to capture and use that data.

Always assume your data will be stolen and use proper data encryption

In June, 2015, a massive breach of US government data was said to expose the personal information on millions of government workers, since the data itself was not encrypted (source: Computer World). No matter how big you are, you should always assume that the possibility exists that your database security will be breached, and data stolen. All sensitive information should always be properly encrypted.

Let's drill down into some of these issues a bit further to see the cause and effect of the choices we make as users and developers.

Poor Password Choices

As we stated above, users are notorious for choosing highly unsecure passwords for their accounts. To expand on that proof point, let's look at the top passwords of 2015, compiled by SplashData from files containing millions of stolen passwords that have been posted online during the previous year.

Table 1-1. Most popular passwords of 2015

1: 123456	6: 123456789	11: welcome	16: dragon	21: princess
2: password	7: football	12: 1234567890	17: master	22: qwertyuiop
3: 12345678	8: 1234	13: abc123	18: monkey	23: solo
4: qwerty	9: 1234567	14: 111111	19: letmein	24: passw0rd
5: 12345	10: baseball	15: 1qaz2wsx	20: login	25: starwars

Before we get too far up in arms about people choosing these passwords, there are many issues with the data sets that we need to be aware of:

- Since most of this data comes from information leaks, it could be that these passwords are just easier to crack through dictionary or brute force attacks.
- We don't know the sources of much of this data, so we can't validate the security measures in place on the sites or services.
- The data may contain anomalies, or simply bad data. If a default password is being set by a service with a lot of leaked data (and never changed), it will push it higher. If we are analyzing data from multiple different sources using information that was poorly parsed, or has those anomalies, the list will be skewed.

With that said, even though those passwords may be a smaller number than the lists purport them to be, and the data may be highly skewed, they still exist. What this means, when building a data and identity security system, is that you have to provide an adequate level of protection for these people. Typically, you want to build for the weakest possible authentication system, which, depending on your security requirements, might be comprised of this list.

In many ways this is because of what we expect of people when they are creating a password: provide a password with mixed case, at least one symbol & number, nothing recognizable in a dictionary or guessable from those who know you. These types of expectations create poor usability for the user, where they won't be able to remember the password, and also ensures that they either pick the easiest way they can to enter the site, or write down that complex password on a post-it note on their display. Usability needs to be a part of identity security for it to be effective.

Security Over Usability

> Favor security too much over the experience and you'll make the website a pain to use.
>
> —Anthony T, *Founder*

Your main objective when handling the data and identity of your users is to ensure their security, but at the same time you don't want to alienate your entire user base by making your sign-in forms complex, or by forcing a multi-screen, manual checkout process for purchasing goods, or by continually challenging the user for identification details as they are trying to use your service. Those are sure-fire ways of ensuring that your users never return.

Main reasons for shopping cart abandonment included the user being uncomfortable with the buying process (it was too complex / lengthy), or they were forced to sign up before purchasing. Many of these concerns can be solved through the usability considerations, such as a single page checkout, and allowing a simplified guest checkout.

The concept of usability versus security is always a balancing act. You need to ensure that you have a high enough confidence in the security of your users, and at the same time do as much behind the scenes so that the user isn't forced to break out of the experience of your site to continually verify themselves.

Some of the questions that we can ask ourselves, when thinking this through, are:

- Can I obtain identity information to increase my confidence that the user is who they say they are, without imposing additional security checks on the user?
- If I have a high confidence that the user is who they say they are, can I build a more usable experience for that user versus one that I have no confidence in?
- What content requires user identification, and when should I impose additional levels of security to verify that?

We'll explore these concepts further in Chapter 3, as we learn about trust zones and establishing identity information on a user.

Improper Data Encryption

Data security and identification isn't about planning for the best, it's about planning for the worst. If there is the possibility of something happening, you should assume that it will happen and have a plan in place to decrease or mitigate the damage that is done.

On March 27th, 2015, Slack announced that their systems had been breached, and user information was stolen. The damage of the security incident was lessened because of their strong data encryption methods. From their blog on the incident, "Slack maintains a central user database which includes user names, email addresses, and one-way encrypted (*hashed*) passwords. Slack's hashing function is bcrypt with a randomly generated salt per-password which makes it computationally infeasible that your password could be recreated from the hashed form.". In addition, following this incident, they introduced two factor authentication for users, as well as a password kill switch for team owners that automatically logged out all users, on all devices, and forced them to create a new password.

In the above case, data encryption and quick action prevented a massive theft of user accounts, and lessened the damage to their credibility and the confidence their users had in them. Data encryption isn't always about trying to prevent data from being stolen, it's meant to slow down hackers from decrypting the data long enough to make it either infeasible for them to decrypt massive amounts of data, or delay them until you can take appropriate action.

The weakest link: human beings

As developers and service providers our biggest interest should be treating our users' data with the most respect we can provide. Hence, we try to secure any kind of information a user provides to us by using encryption algorithms, offer safe ways to communicate and continuously harden our infrastructure in an ongoing struggle.

The most important element in this chain, the human being, is often taken out of the equation, and therefore we open up our application towards threats that we might have not even considered when laying out and designing our security layer. Truth is: users tend to go the easy way. This means for us, that people are likely to choose easy-to-remember and short passwords, simple to guess usernames and might not have been educated about current authentication technology like two-factor authentication - also known as 2FA. Two-factor authentication is a technology that we will discuss in depth in the fifth chapter of this book - it certainly deserves extra attention and focus. Also we will discuss a technology deriving from 2FA simply titled n-factor authentication that represents a scalable security approach depending on the use case.

It is easy to understand why people tend to use and especially reuse simple passwords - it saves them time while setting up user profiles and makes authenticating against services and applications a quick to execute task. Especially with the rise of mobile we're often facing small screen estate and touchscreen keyboards which can be experienced as additional burden.

The phenomenon described here is also known as password fatigue. Gladly there are multiple tools that we, as developers, can use in order to counter these problems and ensure a smooth and pleasing registration and authentication flow within our applications while still maintaining user security.

Many operating systems, browsers and third-party applications try to solve the issue of password fatigue by allowing to both generate randomized passwords and by offering a way to store those passwords under protection of a master password.

A popular example would be the password-management application Keychain that got introduced with Mac OS 8.6.

More and more services like 1Password, Dashlane or LastPass offer to generate passwords for their users. This removes the need to come up with a secure password and is often seen as a convenient way to speed up user account registration.

Katie Sherwin, a member of the Nielsen Norman Group, released an article [1] about simplifying password authentication flows and listed these three approaches as a way to improve user experience:

- Show the Rules
- Show the User Input
- Show Strength Meters

By applying these three rules we can ensure that users feel comfortable about the passwords they use and get a clear indication about the password's strength itself. Further research indicates that users that see a strength meter choose more secure passwords - even if the strength indicator is not implemented that well[2].

> Those who saw a meter tended to choose stronger passwords than those who didn't, but the type of meter did not make a significant difference.[3]
>
> —Dinei Florencio, Cormac Herley and Paul C. van Oorschot, *An Administrator's Guide to Internet Password Research*

Single sign-on

Single sign-on, also known as SSO, is a technology that leverages existing user accounts in order to authenticate against various services. The idea behind this concept is prefilling and securing a central user account instead of forcing the user to register at a variety of services over and over again.

Common choices that try to accommodate the wish to reuse user profiles to either provide profile information or to simply authenticate against other services include OpenID, OAuth 1.0, OAuth 2.0 and various hybrid models like OpenID Connect. In

1 *http://www.nngroup.com/articles/password-creation*

2 *http://research.microsoft.com/pubs/227130/WhatsaSysadminToDo.pdf*

3 *http://research.microsoft.com/pubs/227130/WhatsaSysadminToDo.pdf*

Chapter 4 we will focus on a selection of authentication techniques and will discuss both the technical implementation details as well as the security implications.

Understanding Entropy in Password Security

Before we get too far into the weeds, we should first address how we can determine a weak password from a strong one, if that password was created by a human being. The standard industry mechanism for determining password strength is called "information entropy", which is measured in the number of bits of information in a provided source, such as a password.

Typically, if you are using passphrases, a good level of entropy to have at minimum is 36.86 bits, which coincides with the average entropy level of 3 random words selected from a list of 5000 possible unique words from a list.

Password entropy is simply a measurement of how unpredictable a password is. Its measurement is based on a few key characteristics:

- The symbol set that is used.
- The expansion of the symbol set through lowercase / uppercase characters.
- Password length.

Using the information above, password entropy, expressed in bits, is used to predict how difficult it would be for the password to be cracked through guessing, dictionary attacks, brute force cracking, etc.

When you are looking at determining overall password entropy, there are two main ways of generating passwords that we should explore: randomly generated passwords (computer generated), and human selected passwords.

According to one study, titled "A Large-Scale Study of Web Password Habits", by Dinei Florencio and Cormac Herley of Microsoft Research, the entropy level of the average password was estimated to be 40.54 bits.

Entropy in Randomly Selected Passwords

When we look into the entropy of randomly selected passwords (computer generated), the process is fairly straightforward for determining the overall entropy of the passwords, since there is no human, random, element involved. Depending on the symbol set that we choose symbols from, we can build a series of passwords with a desired level of entropy fairly easily.

First, the generally accepted formula that we use to calculate entropy is: $H = log_2\left(b^l\right)$.

Where

- H = The password entropy, measured in bits
- b = The number of possible symbols in the symbol set
- l = The number of symbols in the password (or length)

To come up with the value of b, we can simply choose the symbol set that we are using from the list below.

Table 1-2. Entropy for each symbol in a symbol set

Symbol set name	Number of symbols in set	Entropy per symbol (in bits)
Arabic numerals (0–9)	10	3.322
Hexadecimal numerals (0–9, A-F)	16	4.000
Case insensitive Latin alphabet (a-z or A-Z)	26	4.700
Case insensitive alphanumeric (a-z or A-Z, 0–9)	36	5.170
Case sensitive Latin alphabet (a-z, A-Z)	52	5.700
Case sensitive alphanumeric (a-z, A-Z, 0–9)	62	5.954
All ASCII printable characters	95	6.570
All extended ASCII printable characters	218	7.768
Binary (0-255 or 8 bits or 1 byte)	256	8.000
Diceware word list	7776	12.925

The symbol set you might not be familiar with above is the diceware word list. The method behind diceware is to use a single die (from a pair of dice), and roll it 5 times. The numeric values on the die each time create a 5 digit number (e.g. 46231, matching the value of each individual roll). This number is then used to look up a word from a given word list. There are 7776 possible unique words using this method. See the diceware word list for the complete reference.

Using the formula, length of the password, and numbers of symbols in a given symbol set, you can estimate the bits of entropy from a randomly generated password.

Entropy in Human Selected Passwords

Before we get into measuring entropy levels within a password that was created by a human being, rather than being randomly generated based on security standards, we need to understand these numbers are non trivial. There are many methods that have been proposed for doing so (NIST, Shannon Entropy, Guessing Entropy, etc), but most of these fall short in one way or another.

Shannon Entropy is seen to give an overly optimistic view of password security (while providing no real actionable improvement hints), and NIST a non-accurate (yet conservative) one. Since we always want to err on the side of caution with password security, let's quickly look at the NIST study on how to measure human selected passwords, as that will give us a good starting point.

According to NIST special publication 800-63-2[4], if we take a human selected password, we can measure the assumed entropy with the following steps:

- The entropy of the first character is 4 bits.
- The entropy of the next 7 characters are 2 bits per character (they state that this is "roughly consistent with Shannon's estimate that *when statistical effects extending over not more than 8 letters are considered the entropy is roughly 2.3 bits per character*").
- Character 9 through 20 have an entropy of 1.5 bits per character.
- Characters 21 and above have an entropy of 1 bit per character.
- A 6 bit bonus is given to password rules that **require** both upper case and non-alphabetic characters (This is also a conservative bit estimate, as the NIST publication notes that these special characters will most likely come at the beginning or end of the password, reducing the total search space).
- An additional 6 bit bonus is given to passwords with a length of 1 to 19 characters that follow an extensive dictionary check to ensure the password is not contained within a large dictionary. The reason passwords that are longer than 20 characters do not receive this bonus is because they are assumed to consist of multiple dictionary words placed together into pass-phrases.

Let's take that idea and see what the entropy of a few examples would be:

- **monkey (6 characters) = 14 bits of entropy** (4 bits for the first character, 10 bits for the following 5 characters)

4 *http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf*

- **Monkey1 (7 characters) = 22 bits of entropy** (4 bits for the first character, 12 bits for the following 6 characters, 6 bit bonus for uppercase and non-alphabetic characters being used)
- **tvMD128!Rrsa (12 characters) = 36 bits of entropy** (4 bits for the first character, 14 bits for the following 7 characters, 6 bits for the following 4 characters, 6 bit bonus for uppercase and non-alphabetic characters being used, 6 bit bonus for a non-dictionary string within 1-19 characters)
- **tvMD128!aihdfo#Jh43 (19 characters) = 46.5 bits of entropy** (4 bits for the first character, 14 bits for the following 7 characters, 16.5 bits for the following 11 characters, 6 bit bonus for uppercase and non-alphabetic characters being used, 6 bit bonus for a non-dictionary string within 1-19 characters)
- **tvMD128!aihdfo#Jh432 (20 characters) = 42 bits of entropy** (4 bits for the first character, 14 bits for the following 7 characters, 18 bits for the following 12 characters, 6 bit bonus for uppercase and non-alphabetic characters being used)

You can start to see some holes in the assumptions that the NIST study makes with the last two password examples, where 1 additional character causes the loss of 6 bonus bits of entropy because of the assumption that the password is of significant length that a user would not have chosen a complex string, and that if a string of that length was used for a password, it is most likely several dictionary words put together, such as "treemanicuredonkeytornado", which, based on the NIST study, would actually give us 41 bits of entropy.

As we go further, you can see why determining the security of a human created password can be tricky, and that's because humans are unpredictable. If we plug in a system of security requirements into a computer generated password system, and store that in a password vault application like 1Password, KeePass, or LastPass, then we can have a very predictable environment. That's why, for the most part, we usually take one of two steps (sometimes both) in securing identity in web development:

1. You provide requirements to the user, when they create their password, on how to strengthen their login. This can be requirements for length, non-alphabetic characters, uppercase and lowercase characters, non-dictionary words, etc. For obvious reasons, the usability of this solution is quite bad, and may alienate many users, but the security increases. The problem here is that when we make it harder to create a password, it is more likely that a user will forget that password, then requiring the use of the "forgot your password" reset flow.
2. You attempt to harden the data, as best you can, behind the scenes. This usually involves encryption, salting, and key stretching (all concepts we will dive into in Chapter 2), to try to help prevent weak passwords that are stolen from being compromised. When you have a solution like this, you may also see a mechanism that only allows a certain number of login attempts before temporarily locking

the account, to prevent potential brute force attacks against weak passwords. This solution is higher on the usability side, since users can practically pick any password they want, but lowers the overall security of their account.

In the end, we're back to questions of usability versus security, and the truth of the matter is that our ideal scenario, for all parties, is somewhere in between. Remember, the two aren't mutually exclusive.

Breaking Down System Usage of a Username and Password

Another important step in understanding the concept of a username and password is to break down what they represent in an identification system. If we put this simply, they are an identification of who you are (the username, or public key) and then a verification of that fact with something that only you should know (the password, or private key).

With that understanding in place, there are two ways that we can think about handling data in an authentication system:

- **Harden the System**: This is the case where we take an existing (or new) system that is built on top of a traditional username and password, and attempt to strengthen it.
- **Remove the Username and Password**: In new or innovative technology solutions, this is the case where we apply the concepts of a username and password, but do so in a different way.

As you dive further into each chapter, our main goals will be to build upon these two concepts, focusing on hardening the system, or finding a new methodology for building our identity and data security with new tools and techniques.

Securing our current standards for identity

Enhancing the security of an existing system is usually the choice of the vast majority of us, as we are building on top of existing work, or we're building a product where the use of a username and password is a preferred login mechanism for the users.

As we explored earlier in this chapter, users are usually the worst people to put in charge of protecting their own security, through their passwords. The vast majority of the population will choose passwords that they can remember, which is almost always the complete opposite of what we would traditionally think of as a secure password.

What we know from earlier sections is how to approximate the predictability of a password, and that we should always build our security towards the most unsecure element in the chain, not the average.

With that said, there are certain standard mechanisms that we use for account security, and others that we should avoid.

Good and bad security algorithms

Not all encryption algorithms are created equal when it comes to the security of our data and privileged user information. Some are built for speed, for quickly and accurately encrypting and decrypting large amounts of data, and then there are others that are designed to be slow. Let's say your database of a million encrypted user records has been stolen, and the attacker is attempting to crack the encryption, such as by trying every word in the dictionary as an option, to reveal the data underneath. Would you prefer to make this as fast as possible for them to check different words, or as slow as possible? The correct answer is that you want this process to be as slow as possible for them.

With regular cryptographic hash functions, an attacker can guess billions of passwords per second. With password security hashing algorithms, depending on what the configuration is, the attacker may only be able to guess a few thousand passwords per second, which is a massive difference.

The Good

These are hashing algorithms that are meant to be used for password security, and are built to be purposefully slow to make cracking the data harder.

- **PBKDF2**: PBKDF2 stands for "Password-Based Key Derivation Function 2", and was created by RSA Laboratories. It applies a pseudorandom function, such as a hash, cipher, or HMAC, to the input (password) along with a salt. The process is repeated many times, which produces a derived key.
- **bcrypt**: Created by Niels Provos and David Mazières, bcrypt stands for "Belgian Fundamental Research in Cryptology and Information Security". It is a key derivation function based on the blowfish cipher. It incorporates a salt into the process to protect the key, and also has an interesting adaptive functionality to it. Over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force attacks.
- **scrypt**: Created by Colin Percival, scrypt is another key derivation function that is designed to combat large-scale hardware attacks by requiring high amounts of memory and therefore slowing down computation.

The Bad (for passwords)

These are our standard cryptographic hashing algorithms, which are meant to be fast. In the case of password security, this is not a good scenario as slowing down the algorithm makes it much harder for an attacker to crack the data.

- **MD5**: MD5, or "Message-digest algorithm", was designed by Robert Rivest in 1991, and produces a 128-bit hash value, typically expressed as a 32 digit long hexadecimal number.
- **SHA-1**: SHA stands for "Secure Hash Algorithm". It was designed by the NSA, SHA-1 produces a 160-bit (20-byte) hash value. This hash value is typically rendered as a 40 digit long hexadecimal number.
- **SHA-2**: Also designed by the NSA, SHA-2 is the successor of SHA-1, and consists of six hash functions with hash values that are 224, 256, 384 or 512 bits (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256).

What data should be protected?

We've hinted at this one a few times during this chapter, but when it comes to asking yourself "What information absolutely needs to be encrypted?", the answer is actually pretty simple: Anything that is personally identifiable (identity data, personal information, payment details), or anything that is imperative to your system, that could open up additional leaks or holes in your architecture if released.

Account Recovery Mechanisms & Social Engineering

After we've reviewed the details worth protecting, we should take this knowledge into account when looking at recovery mechanisms. Often Social Engineering or weak recovery mechanisms lead to exposure of information - even though protection mechanisms got implemented in order to prevent exactly this. If you are familiar with these matters, feel free to skip to this chapter's wrap-up.

Popular examples include customer support providing account details they're not supposed to share, and badly-planned password-reset flows. A compromised email account can lead to easy access to a user's account - securing our users by offering sensible security questions and allowing to provide specific responses can help lowering the risk of information leaks.

> Social engineering is a non-technical method of intrusion hackers use that relies heavily on human interaction and often involves tricking people into breaking normal security procedures. It is one of the greatest threats that organizations today encounter.
> [5]

[5] *http://searchsecurity.techtarget.com/definition/social-engineering*

—TechTarget SearchSecurity

The Problem with Security Questions

While the overall knowledge and consciousness about secure passwords is steadily growing, another volatile area - Security Questions - is often ignored. Instead of offering the user an array of personal questions or even allowing for the definition of your own security questions, a number of generic phrases is offered that is often as easy to find out as searching for a person's social media profile.

Security Questions often appear as repetitive and sometimes even inadvertently comedic collections that can be cumbersome to answer and hard to remember ("What was my favorite dish as a child?", "What's your favorite book?"). Soheil Rezayazdi published a list of Nihilistic Security Questions on McSweeney's Internet Tendency[6] that should at least cause a slight smile on your face - here are our personal top 5:

1. When did you stop trying?
2. In what year did you abandon your dreams?
3. At what age did your childhood pet run away?
4. What was the name of your favorite unpaid internship?
5. What is the name of your least favorite child?

In all seriousness, the impact of Social Engineering is often completely underestimated or even ignored. It is often easier to pass barriers instead of circumventing and breaking them down. The scope of Social Engineering can be anything between looking up some facts about a person online and sneaking into office buildings; while this might sound like an exaggeration (and often does not have to happen), it makes sense to prepare and train staff accordingly.

In case you are looking for more information on this topic: great resources on Social Engineering are Kevin Mitnick's books "Ghost in the Wires", "The Art of Intrusion" and "The Art of Deception"[7].

6 *http://www.mcsweeneys.net/articles/nihilistic-password-security-questions*

7 Kevin Mitnick rose to mainstream fame by hacking companies such as Nokia and Pacific Bell. He's nowadays active as a security consultant.

Next up

Now that we understand all of the concepts that we are going to be using and talking about throughout the rest of the chapters, let's jump into the next chapter by drilling down into how hashing, salting, and data encryption can be added to your systems.

CHAPTER 2

Password Encryption, Hashing, and Salting

Jonathan LeBlanc

In the first chapter we learned about the underlying concepts of password security, and the current state of the industry and standards employed. Let's start putting some of that into practice as we explore the practical application of password encryption and security.

To start this implementers approach, let's first look at the different ways that data will be transmitted and stored, that we should be aware of.

Data at Rest vs Data in Motion

As we start to explore the concepts of data security, there are two important concepts that we should adress: data in motion versus data at rest.

When we talk about data at rest, what we actually mean is the inactive (or resting) digital data that is being stored on your servers, such as the databases that you are using to store passwords, profile information, or any other details needed within your application.

When we discuss the concept of data in motion, what we're talking about is any data that is in transit, being sent back and forth from an application to a database, or communication back and forth between websites and APIs or external data sources.

Data at Rest

> If you're talking about credit card environments, where you've got a requirement to encrypt the credit card information at rest, I think the most common method people

use there is enabling encryption within the database. That's typically about as good as it gets in terms of host-based encryption[1]

—Chris Gatford, *Hacklabs*

When it comes to web and application developers, the concept of protecting the database in which we store secure information about our clients is typically something that we rarely have to encounter, but it is a concept that should be understood. While the aspects of data at rest are beyond the scope of this book, let's cover some of the basic concepts and guidelines to understand that database encryption is absolutely needed, even though in 99% of organizations, this is simply not done.

If we've tried to reiterate on a few occassions, you should always assume a worst case scenario when planning for data breaches. In this case, we should assume that an attacker has gained access to our database, with the end goal of capturing any senstive data and passwords. Wouldn't you want to have both the password encryption to prevent account access, as well as an additional layer of encryption on the database itself?

First, let's address the encryption methods that should be used on the database. Different to the standards that we discussed in Chapter 1 for password encryption, the strong encryption methods that should be used for database encryption are SHA-256 (Secure Hash Algorithm) or better, AES (Advanced Encryption Standard)), and RSA (Rivest-Shamir-Adleman)). These are all part of the NIST approved algorithms. Weak encryption algorithms, such as MD5 and SHA1, should never be used for database encryption.

Now, there are a few standards that should be followed:

- Keep access control (user login) separate from database encryption. Should a username or password fail, the database itself should remain encrypted, effectively providing multiple levels of protection.
- The keys that are used for the database encryption should be updated on a regular basis.
- Encryption keys should always be stored separately from the data.

Data federation is another method to help prevent unwanted access in the case of an application with global reach and data storage. The purpose of this strategy is to maintain distinct database systems in the regions where the personal information is actually needed (e.x. the personal information of a UK customer is stored within a database in the UK, not a centralized database in the US). Effectively, instead of having a centralized database with all customer information that is copied around to data

1 *http://www.zdnet.com/article/encrypting-data-at-rest-is-vital-but-its-just-not-happening/*

centers as needed, only information in the region that it is needed in is maintained. This type of strategy is effective when government regulations / laws require access to be granted to all user information that is stored in their country, regardless of whether that data belongs to individuals in other countries.

Lastly, there is one underlying concept that should be understood and implemented. You should only ever store the minimum amount of sensitive user data that is required to run your application, site, or services. A major industry trend over the last few years has been to capture as much information about your users as possible, store it, and then figure out whether it is useful and is viable at a later date. This is **absolutely not** the practice that should be employed when you are trying to provide the best level of protection for your users. When creating the architectural schema for your application or website, you should include think heavily on the type of data that is needed for both the state of the application, and use that to build the database structure for personal information that should be stored. The less priviledged information stored, the less potential impact on your customers.

Beyond the user data, sensitive financial information such as credit card data can also be offloaded, typically to the payment provider through a system such as a credit card vault. In addition to the security benefits in not having to host that data yourself, you don't incur the implications of having to implement all standards for PCI DSS compliance, as required when hosting payment information for customers.

Data in Motion

Data in motion, or data that is in transit, is what the vast majority of web and application developers will be dealing with in their day to day development. Realistically, this will encompass a number of different scenarios, including:

- Signup information from a user that will be used for account access and identity.
- Transmission of profile information to and from service APIs.
- Other data collected through the application or website and transmitted for database storage.

This is the data focus that we will be exploring throughout the upcoming chapters. Our first step is to look into security and encryption behind the user profile, through the proper storage and use of the user password.

Password Attack Vectors

There are many methods in which an attacker may attempt to gain access to the account of a user. Some of which are geared around manipulation of the user themselves, while others attempt to target the application or website itself to gain access.

A few of these are:

- **Phishing:** This is the concept behind tricking the user into providing their login credentials through a malicious site or application. Typically you would see these types of attempts come through email scams, where the sender portrays to be the company in question, and requires the user to log in to the malicious site for some reason, thereby stealing their login credentials, and access to their accounts.
- **Social Engineering:** Taking the concept behind phishing to a new level, social engineering hacks are usually orchestrated by other communication means, such as through phone calls. The attacker pretends to be a network technician, or some sort of IT security for a company, and asks the user for their login credentials to repair the issue that they are calling about. In doing so, they gain access to the user account.

As you can well imagine, it is difficult to build a safety net for cases such as these, but when it comes to attacks against the website or application that we are working with, that is an area where we definitely have the control to build safety measures into our login controls, profile systems, and database structures.

These attack vectors include:

- **Brute force attacks:** Calculate every possible key variation within a given length, and then trying each one, one after another, until the password is guessed. The shorter the password, the quicker this method takes to work. As the length of the password increases, the time to crack the password increases exponentially. When this method becomes too costly, other methods, such as dictionary attacks, are employed. One of the methods employed to counter brute force attacks is key stretching, which we'll explore later in this chapter.
- **Dictionary attacks:** The premise behind this is to loop through a list of predetermined words / terms, such as all words in a dictionary (hence the term "dictionary attack"), and trying all combinations against an encrypted password to find matching values. Unlike a brute force attack, a dictionary attack is just trying input that is considered to, most likely, be a common input word, term, or phrase. Use of a salt (which we'll discuss below) is an appropriate way of dealing with these types of attacks.
- **Rainbow tables:** Essentially, rainbow tables are large lists that contain precalculated hashes (for a given hash function), and the passwords from which they were derived. Unlike a dictionary attack where the hash function is applied at each crack attempt, the attacker can simply compare the pre-calculated hash with the password hash from the user database, making the entire process more efficient. Use of a salt is an appropriate way of dealing with these types of attacks.

This type of attack is typically carried out offline, when the attacker has full access to the data.

- **Malware:** These are key loggers or screen scrapers that might be present on a users machine, logging activity during log in or signup. Typically, these would be used in conjunction with social engineering to prompt the user to load or install a piece of infected content. An appropriate way of dealing with these attacks is to use a second factor of authentication (e.g. text message verification) during login.
- **Offline cracking:** We cannot forget that there's always the possibility that all of our stored user information will be stolen, giving an attacker all the time they want to crack a password hash. Once the data is stolen, hardening techniques such as hashing our passwords with a salt, and using appropriate hashing mechanisms that are built to slow down cracking, are the barriers to data theft.

With an understanding of the general landscape of attack vectors, let's dig into some of these in more depth.

Brute Force Attack

Brute force attacks, also known as an exhaustive key search, is the practice of attempting to break password encryption by going through all potential permutations of a password for a given length. In other words, let's say that we're trying to break a password of a known length, up to 5 characters, a brute force attack will attempt every possible password permutation of that length.

Obviously, this is not the first method that an attacker would use, due to the length of time that it would take to crack a password. For a password with a key length of **N bits**, the time that it would take to crack the password will be proportional to the number of bits, which would be proportional to 2^N in the worst case, but half of that on average. Thusly, as the password length increases, the time to break the password also increases, exponentially.

Other methods, such as the use of rainbow tables, or dictionary attacks, are more viable attack vectors. Typically brute force attacks will only be used with offline data (not a direct site attack, but data that was downloaded in a hack), and will only be used when other more viable vectors are unavailable to the attacker. If proper password encryption methods are employed that utilize key stretching, this attack method becomes incredibly negligable.

Given these facts, when it does come to securing your web application from potential brute force attacks, there are a number of methods that can be implemented to prevent this attack vector, including:

- Implementing a CAPTCHA (Stands for *C*ompletely *A*utomated *P*ublic *T*uring test to tell *C*omputers and *H*umans *A*part)[2]) following an unsuccessful login attempt to increase login complexity and help prevent automated attacks.
- Adding in a 2FA (Two-Factor Authentication) verification mechanism, such as through an SMS to a verified phone number using Authy[3]) or some other like service.

We'll go through an implementation of 2FA with Authy in Chapter 5, but let's see what a CAPTCHA implementation might look like when integrated within our site.

Creating a CAPTCHA with reCAPTCHA

As we talked about earlier, one of the methods that can be employed for preventing an attacker from attempting password after password for an account is to place a CAPTCHA in front of them. Typically you'll want to make use of a CAPTCHA only after one to two failed password attempts. These may be a user who has simply forgotten their password, or they may be an attacker attempting different password combinations in an attempted brute force attack.

Using a CAPTCHA

As with most security precautions, you want to try to impact users as little as possible. This is the reason that we don't want to show a CAPTCHA for every login attempt, as it's just an extra step that a user has to go through to log in. When restricting use to only after one to two failed attempts, you ensure that the vast majority of your users are not impacted, and those that are will mostly understand the reasons for the added security on successive password attempts.

One of the leading CAPTCHA systems is reCAPTCHA by Google[4]. The simply interface typically will involve just clicking a box to confirm that you are "not a robot". It provides a very nice and simple user interface so that your users aren't heavily impacted like with traditional CAPTCHA systems that require you to type in words from a picture or audio recording, while still providing high levels of security potential bots and automated attacks.

2 *https://en.wikipedia.org/wiki/CAPTCHA*

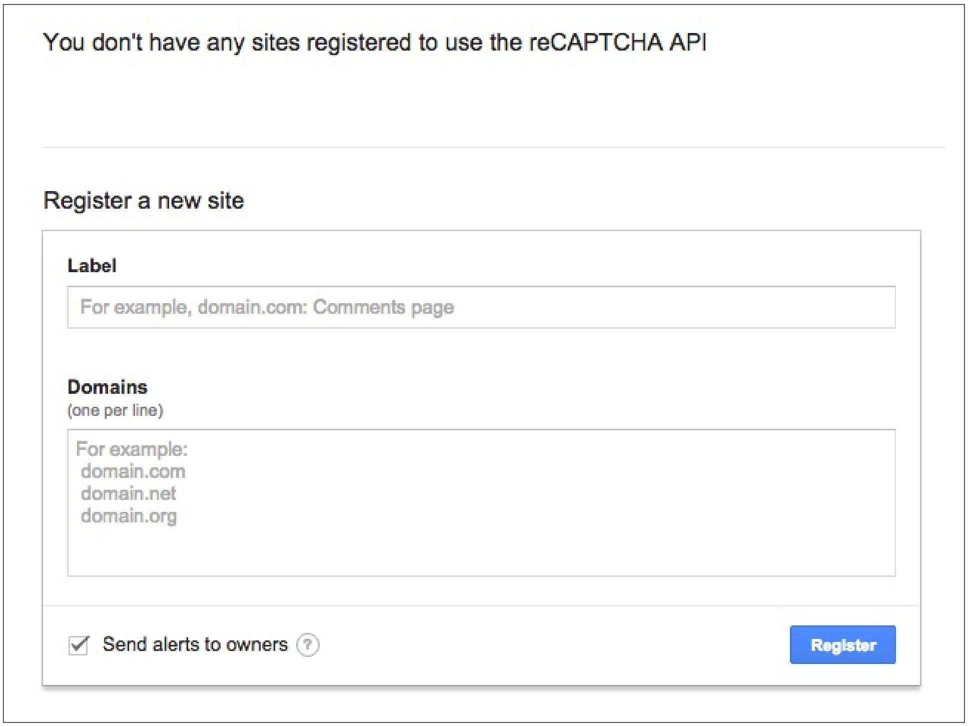
3 *https://www.authy.com/*

4 *https://www.google.com/recaptcha/intro/index.html*

With that said, let's see how we're going to implement reCAPTCHA on one of our sites, and then confirm user input on our Node server.

The first step is to head over to the reCAPTCHA admin page to sign up for an API key pair for your site, that will allow you to use the system.

On this page (given that you don't have any existing keys), you will be met with the registration system that looks like the image below:


You don't have any sites registered to use the reCAPTCHA API

Register a new site

Label

For example, domain.com: Comments page

Domains
(one per line)

For example:
domain.com
domain.net
domain.org

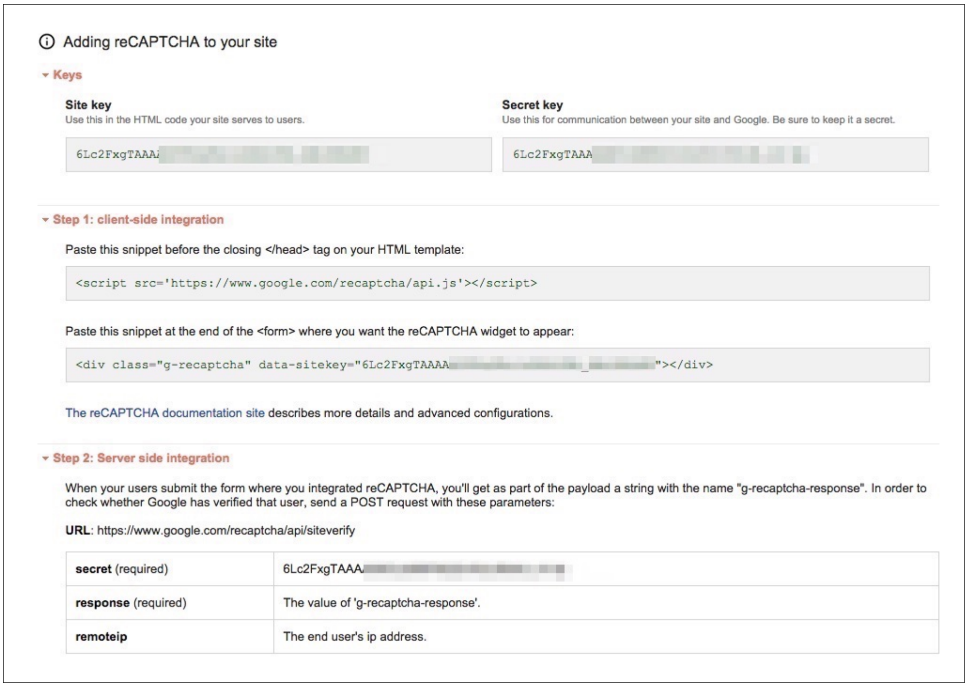
[✓] Send alerts to owners

Figure 2-1. Registering your Keys

Following the instructions on the page to register a new site, we enter a few pieces of information:

- **Label**: Something to identify the site or keys for yourself, such as the site name or URL.
- **Domains**: The root domains that you will be placing reCAPTCHA on. This can be multiple domains that you maintain.

When you click the register button, you will now see all of the setup information that you will need to go through to integrate reCAPTCHA on your site, as shown below:


ⓘ Adding reCAPTCHA to your site

▾ Keys

Site key
Use this in the HTML code your site serves to users.

```
6Lc2FxgTAAA[redacted]
```

Secret key
Use this for communication between your site and Google. Be sure to keep it a secret.

```
6Lc2FxgTAAA[redacted]
```

▾ Step 1: client-side integration

Paste this snippet before the closing </head> tag on your HTML template:

```
<script src='https://www.google.com/recaptcha/api.js'></script>
```

Paste this snippet at the end of the <form> where you want the reCAPTCHA widget to appear:

```
<div class="g-recaptcha" data-sitekey="6Lc2FxgTAAAA[redacted]"></div>
```

The reCAPTCHA documentation site describes more details and advanced configurations.

▾ Step 2: Server side integration

When your users submit the form where you integrated reCAPTCHA, you'll get as part of the payload a string with the name "g-recaptcha-response". In order to check whether Google has verified that user, send a POST request with these parameters:

URL: https://www.google.com/recaptcha/api/siteverify

secret (required)	6Lc2FxgTAAA[redacted]
response (required)	The value of 'g-recaptcha-response'.
remoteip	The end user's ip address.

Figure 2-2. Adding reCAPTCHA to your Site

At the top we can see the keys that we will be using for this process. Keep those in mind as we go through the two steps that are needed for integration.

First we need to add the reCAPTCHA script include at the end of the <*head*> section of our HTML document, and create the container for the widget within our login form. Stripping down this process into just the pieces that we need on the page, the HTML code would look like the below:

```html
<html>
<head>
    <script src='https://www.google.com/recaptcha/api.js'></script>
</head>
<body>

<form method="post" action="/capture">
    <div class="g-recaptcha" data-sitekey="6Lc2FxgTAAAXXXXXXXXXXXXX"></div>
    <input type="submit" value="Submit">
</form>

</body>
</html>
```

Going through the above, here's what we're doing:

- The *https://www.google.com/recaptcha/api.js* script include is added right before the closing </head> tag of our HTML document.
- We create a form that will POST the login data to our Node endpoint, /capture (we will set up an Express app to capture POST requests to that endpoint).
- In the form, we simply have the <div> that will house the reCAPTCHA widget (placed where I want it to show up in the form), and a submit button to submit the form to our server-side endpoint.

When we load that form, we should see something like the below on our page, clearly showing the reCAPTCHA widget and our submit button.

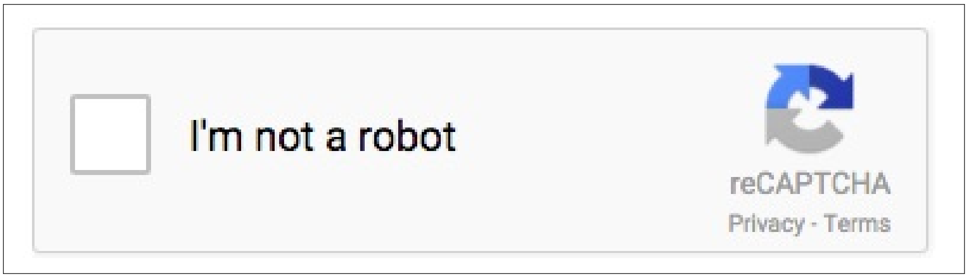


Figure 2-3. reCAPTCHA on your Site

The user will interact with the widget and click (or not) the option to identify themselves as human. When the form is submitted, the POST data will be sent to our /capture endpoint. For the reCAPTCHA component, the data the we will need to capture for the verification step will be located within the **g-recaptcha-response** parameter, and look something like the below:

```
g-recaptcha-response=03AHJ_VusWHPddH611975sAE4rH9twXhs05HZcIlUa4Yv9eczU_aFxLl2VeB
kisEkZdCBd7a1R35rNvCikbsgkAsEN8KoP400SEFhzNebZN3yaK4826QQT2W0jjaK-fGIVxWGiTzHrcBW
roHDAbImEpukdJj58yN_vJFsgrnSvmXV3jWK09f_zqiOpOw07V848yYnXnIQdCuqR3SKJEvexIEmlRewZ
GnJvnN2pKMaQ_Hcnjp5p2mc5Nm-z_bELGwf2isrQvw8zm9m4lA9Etu0CS-N3PwZ_R0kELWdSTwNYH7aI8
wlWMHct8A71LDy_t82maP0jC07y6sVzlRJLQ5dsJ5gXCpnuUDPTfeASoJosTUChPPXjYWDEiZ8dAJxxNP
SNdyhftEXtrN7PiebkIEVngwRxVUqZRLe9JQpLk9HimOoOuuu5gKva4Ai_-ohHHqaAem6e_AJe6GnWO8f
PDpqXBcNOU_kkDOfQ_zHZ7FVoOvhbBW8GwV5xRjOB-7yxibHguemcm2X22W2atX0TC_hIaJZjWYZncGms
3Hgeq5lLTKzInV1W6kHmCvGqCVrakhHjQn1tfQppXsPZyWPx6RWzNKR0Mloe8bYefx2VdYZAhXeJfDMSD
sq1c7KYGJctNXiL34QIGwWxyPkUCYUxMHACkGyryRCxbYKDwi6fdsONeQHe4nhGwFneKU4kI4Kp-ymEgc
HvDUaTGS8sLrXiY36eKZrB6CIS0P4pQbJr4TJvt2dE9VkVPjKsyvRKMONpIu6G0pJsxb0ssUHHa_iTK7w
s0k681LM7LKH_MxtQJKwl8_6HycyhDn-BAjx8YEZ-KSslUvuVSelSxEo1R_y_n5MGo-qrRVSmKjP14O2k
DBF0vlW1UZTIgl2gc6Iz_QU6Oz6JQOUJOAZNtkMv6aWu5h-uVkMcIRqIHWWFqg
```

At this point we need to set up that /capture endpoint to accept the form POST from our HTML document. We'll be using Express for this one, with mostly standard packages. Since we will be dealing with JSON responses, the only package that we'll need to pull down from NPM is body-parser, to handle those responses. Start out by installing this via the following terminal command:

```
npm install body-parser --save
```

Next, let's set up the variables for the packages that we'll need, as well as the configuration for body-parser to handle JSON data responses.

```
var querystring = require('querystring'),
    bodyParser = require('body-parser'),
    https = require('https'),
    app = require('express')();

//support JSON & URL encoded bodies
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));
```

Here is what each package above will be used for:

- **querystring**: Converting JSON objects into strings for POSTing.
- **body-parser**: Accepting JSON responses from the verification step.
- **https**: Making HTTPS requests to verify the reCAPTCHA data sent from the previous form.
- **app**: Express

Now let's build an app handler for the data that will be POSTed from our previous form to the /capture endpoint. That block will look like the below:

```
//handle all POST requests
app.post('/capture', function (req, res){
    var response = req.body['g-recaptcha-response'];

    var verify_data = querystring.stringify({
        'secret' : 'YOUR SECRET KEY',
            'response': response
        });

        //uber access token fetch endpoint
        var verify_options = {
            host: 'google.com',
            path: '/recaptcha/api/siteverify',
            method: 'POST',
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
                'Content-Length': verify_data.length
            }
        };

        //set up request
        var post_req = https.request(verify_options, function(result){
            result.setEncoding('utf8');
```

```
                result.on('data', function (verification){
                            console.log(verification);
                });
            });

            //post data
            post_req.write(verify_data);
            post_req.end();
    });
```

The above will run for all POSTed data sent to the /capture endpoint. When data arrives, we run it through several steps:

- We capture the reCAPTCHA data to be verified within the POSTed data, located within *req.body[g-recaptcha-response]*.
- Next we build the POST object that will be needed to send to the verification endpoint. The data required will be your secret (this is the key that was given to you at the beginning of this section when you signed up for your keys) and the response from the POST body.
- We then build out the endpoint data for the verification step. The endpoint to send this data is *https://www.google.com/recaptcha/api/siteverify*, so we set the host to *google.com*, the path to */recaptcha/api/siteverify*, and the method should be POST.
- Next we set up the request. Since this is an HTTPS endpoint, we need to use **https.request(...)**, passing along the endpoint option variable that we just created. When the results from that come back we will simply be logging out the response.
- Lastly, we begin the request by sending through the verification data variable that we set up at the beginning.

The last line is to start our server.

```
app.listen(process.env.PORT || 3000);
```

In the case of testing on localhost, when we run the above it'll listen on localhost, port 3000.

At this point the verification has been sent, and a response on whether this is a human or not (that the person clicked the "I'm not a robot" box) will be logged out from our above code.

If successful, the data response that we will see will look like the below:

```
{
  "success": true
}
```

If not successful, we'll see a response like the below:

```
{
  "success": false,
  "error-codes": [
    "missing-input-response"
  ]
}
```

Basically, if we read the success paramter from the response, we can see whether we should process the login or not.

Using this method, we can prevent an automated script from simply making request after request to our login for a given username, passing through every possible variation of a password that they can generate, leading to an account being compromised.

Dictionary Attacks

A dictionary attack is slight different than a brute force attack. Instead of trying every permutation of a given password length, a dictionary attack takes words from a prearranged list (typically all words in a dictionary and other common password choices), encrypts those words, and compares the encrypted string against the encrypted password obtained from a user's account.

In a practical sense, let's say that we had each of the words that we wanted to check, such as the following (as just an example):

```
var words = ['animal', 'monkey', 'test', 'tornado', 'mango', 'slide', 'pepper',
             'diaper', 'tube', 'cloth', 'hair', 'smell', 'eyes', 'tip', 'right',
             'wrong', 'happy'];
```

Now, to conduct this type of attack we would encrypt those words, perhaps with an obtained salt (a random unique string used to strengthen weak passwords) from a data breach, or perhaps without a salt, to give us a hash to compare. Given the words, we might now have a list of hashes that appear as such:

Table 2-1. Dictionary words and associated hashes

Common Word	Associated Hash
test	$2a$10$wkmirMIMsQxpSvKpn.KNyOTU65YuneDcMXwA7XEHR5brjhpjugWCm
animal	$2a$10$.hfSmZVMv3kv5SE9hMun805p/AAWsH7eOrhfYdItG0UCaU/aRT85W
tornado	$2a$10$GI5itVWvmom3vBLTCSsZJ.vUtp5qzAnjVUK5PG4PRIbleIkw8BIdC
monkey	$2a$10$mmKLHtnYIlvGrcwF9pXgjOEkczGm99f/iNU3qCA2GOySPgOjAeguO
pepper	$2a$10$wbNHHKyHhIhToY6kpExOZO.qIS4UURMm7sKAUTLudiiyMO8wD.HGq
slide	$2a$10$/j9vzCZqmGvhGLMUFhwJ2.JvsiZ3i/MsXTfsf9VItR8Gitg.GWgv2
mango	$2a$10$TFKgAYZrfb7p/J6Mz1NZsuhlp62Sa24GKBb7G8q4j702rc1Ntlopa
diaper	$2a$10$/ISCPiQC5wGh4JF9bXKJmeI9OKvWdikq8cUuKmXvh5Za9HWcOADVy
tube	$2a$10$lkZSe0Y1h710V4JGV5hBD.ZVttOnizitpqdeu1nQvO7txdemlvta2
cloth	$2a$10$et.LBm.NeYWXhVl/TFB3cOs0kRAPFh8iBjSicap1ZUYNBqFIOLUp2

Common Word	Associated Hash
hair	$2a$10$6taHB2eQJDLeUUYL7Fw.0.u1avLILk0t74Jhv1uBHv350QAvwKRgC
smell	$2a$10$nVJwlzP5yheet0a8ALQGBehoreNsfY7eyC4X76tl3ZdiCGYtHNg4m
tip	$2a$10$xeKdWclook9IOKjcQO2GkOpzgPo.pkbc3QVIFsGKfv6UqYV2KoZIG
right	$2a$10$Y0pfFl08OWinGu1/1T7NHe6LsE1ey9ggq7.klvoiS2jkzSMpjZfZu
wrong	$2a$10$6Q34ws6flQDvZU6RftuaJeWC40c8GCO2NeZfmCHyoW7aZv9H1sYG2
eyes	$2a$10$vlcnd/G9fyDYVklgvRhTUuTw26L57nw4MuZEYqHv2dSYiyppCnbA.
happy	$2a$10$3c3lF6ALH4kab4Cd8Zeq50JEfSF9EcOcVJlxL5Ra.x9g8OVCjKKti

From our data dump, let's say the user record has an encrypted password of **$2a $10$TFKgAYZrfb7p/J6Mz1NZsuhlp62Sa24GKBb7G8q4j702rc1Ntlopa**. We compare all of the hashes that we have produced above against that user hash, and find a match with the word **mango**. We now have a cracked password that we can use to gain access to the user account.

The way we typically protect against dictionary attacks is through employing a salt in your password encryption. Using a salt means that the attacker cannot just employ a standard encryption algorithm to generate the hash, but they also need to compute the hash with the associated salt.

Reverse Lookup Tables

Taking the process of dictionary attacks a bit further, and thus reducing time to decrypt, what a reverse table basically does is store the plaintext variation of a password beside the associated hash of the password in a table. The table is stored to be searched on the hash as opposed to the plaintext password.

If we go back to our dictionary attack list, it would look very similar:

Table 2-2. Example lookup table

Hash	Associated Word
$2a$10$wkmirMIMsQxpSvKpn.KNyOTU65YuneDcMXwA7XEHR5brjhpjugWCm	test
$2a$10$6Q34ws6flQDvZU6RftuaJeWC40c8GCO2NeZfmCHyoW7aZv9H1sYG2	wrong

You start with obtaining the hashed password from the user record. You then compare that hash to the other hashes in your database. If there is a match, you pull the associated plaintext word for the password.

Like dictionary attacks, salting a password during hashing makes reverse lookup tables essentially useless. Even if the salts were obtained in a user information data breach, a lookup table would need to be generated for each word with the associated one time use salt, making it incredibly inefficient.

Rainbow Tables

There is typically much confusion over the difference between a rainbow table and a simple lookup table that stores a hash to a matching plaintext password. Rainbow tables are essentially a way of reducing the amount of storage needed to calculate a long list of password guesses to try in order to break a hash.

Let's compare that space / time tradeoff. A rainbow table attack takes less processing time than a brute force attack, but uses more storage. On the flip side, a rainbow tables needs more processing time than a simple lookup table, but requires far less memory.

There is one important aspect that is required for rainbow tables to function, and that's called a reduction function. In short, the purpose of a reduction function is to take a given hash and run the algorithm to generate the next possible plaintext password for us.

For instance, say the passwords that we are looking for are simply numeric, and a maximum of 5 digits in length. Our reduction function can simply have an algorithm that pulls the first 5 digits from the resulting hash, like so:

1. We start with arbitrary password 12345.
2. We use bcrypt to hash that password, giving us a hash of $2a$06$qEMn/vmty3PCCc5qxyOpOOjbJYnokP9zfwWVxT1jnfJqIQwOzuqjq.
3. We use our reduction function to take the first 5 characters we find in the hash.
4. That gives us the next plaintext password to try, 20635 (the literal first 5 numbers we encounter in the hash).

Here's how this whole process works in detail. Let's say we want to generate 10,000 potential plaintext passwords, and their associated hashes, to compare against hashes that we have from a compromised list of user records. If I was using a lookup table, I would have each of those plaintext passwords mapped directly to its hash. That requires a lot of storage, but is rather trivial to query to see if I have a match.

Here's how a rainbow table stores a chain of plaintext passwords and their associated hashes.:

1. We take some arbitrary password, such as treehouse.
2. We then hash that, say with bcrypt, to give us a resulting hash, $2a$06$TjlWuN71X8GshO31hK8qVueHhV4nsTi9ZGxk9fBSxwiU49nBw8kVy.
3. We then run that hash through our reduction function, giving us our next viable plaintext password.
4. Next, we repeat steps 2-3 for a lengthy number of chains, say 10,000.

5. Here's the secret. We only store the first plaintext password, and the last hash in the 10,000 word/hash chain. What we have essentially done is created a list of 10,000 plaintext / hash pair guesses, while only storing one plaintext password, and one hash.

> A rainbow table is not a decoding system for a hash, as hashing is built to be one-way (can encode, but not decode). A hashing function allows you to map a plaintext password to a resulting hash, but if you try to get a plaintext password back from a hash, you'll only get some other random plaintext password. A rainbow table works in reverse, mapping a hash to its associated plaintext password. We aren't docoding, we're mapping.

Let's say we now have a hash that we want to get the plaintext password for. We follow a few steps to try to get it:

1. First we look through the list of hashes that we have stored in our table (that last hash of a 10,000 word/hash chain). If we find a match, we just grab the associated word for that hash that we already precalculate.
2. If there is no match, we move to the next hash in the chain, hash #9,999, and do the same thing.
3. We then follow this process all the way to the end of the chain, tryng to find the associated plaintext password.

In a realistic implementation, we would have a multitude of chains created that we could run through. With multiple machines, we would run these chains in parallel to reduce the amount of time it would take to process the attack.

The best way to combat this attack is, again, through salting. In the case of attack vectors like this, having a long, complex password becomes important because:

- It takes exponentially more time to run these attacks with each additional character added.
- Parrot and parrot (change of case) need to be stored as different attack cases in the table, as they contain different characters. Adding mixed cases and special character through the salt allows you to increase the size of the character set, and thus potential guesses, that an attacker needs to run through.

With that said, let's look at the process of salting in more depth to truly understand how to properly implement it in our hashing functions.

Salting

A salt is some sort of random data that is used in conjunction with the user password, when hashing, to harden the data and to protect against a few of our attack vectors, specifically dictionary attacks and rainbow tables. By providing that piece of random data, of significant length, what you're essentially doing is ensuring that the produced hash is unique, so even if multiple users have the same password (as we know they do), the unique salt applied to the hash will ensure that the resulting hash itself is unique. The unique hash is what protects us from the hash comparison methodologies behind rainbow tables and dictionary attacks.

Let's look at this in practice. First, let's start by seeing what a hash might look like if we run it through *scrypt* with no applied salt. Let's assume the password that the user is using is "mechagodzilla":

```
//example hashes using the password 'mechagodzilla' and no salt
hash('mechagodzilla') =
    162e0a91026a28f1f2afa11099d1fcbdd9f2e351095ebb196c90e10290ef1227
```

Each time scrypt hashes that password, the resulting hash will remain the same. If "mechagodzilla" is part of the word list tested through a dictionary attack, then it would be an easy matter of comparing the hashes and figuring out the user password.

Now let's see what applying a random salt to the equation will give us. Let's use the same "mechagodzilla" user password, but use a salt generated from the Node crypto library when the hash is created. Here are three instances of that at work:

```
//example hashes using the password 'mechagodzilla' and random salt
hash('mechagodzilla' + '458cf2979ef27397db67077775225334') =
    f3499a916612e285612b32702114751f557a70606c32b54b92de55153d40d3b6
hash('mechagodzilla' + 'ef5b72eff781b09a0784438af742dd6e') =
    7e29c5c48f44755598dec3549155ad66f1af4671091353be4c4d7694d71dc866
hash('mechagodzilla' + 'cc989b105a1c6a5f0fb460e29dd272f3') =
    6dedd3dbb0639e6e00ca0bf6272c141fb741e24925cb7548491479a1df2c215e
```

In simple terms, a salt of sufficient length and randomness provides a massive boost in security towards certain attack vectors, with just that simple, unique addition.

Generating a Random Salt

Let's take a look into how we can generate a random salt for our hash functions using the Node crypto library. Since it is part of the standard library, we don't have to go through the additional step of installing from npm.

The Node crypto library, in addition to providing functionality for generating random salts of varying length, also has built in functionality for working with PBKDF2 to generate required hashes from the user password and salt.

First we start by adding the crypto requirement to our Node project:

```
var crypto = require('crypto');
```

With that in place, we can generate our salt using the `randomBytes(…)` method, like so:

```
crypto.randomBytes(32, function(ex, salt){
    //log readable string version of the salt
    console.log('salt: ' + salt.toString('hex'));

    //proceed to next step: using the salt
});
```

Generating a salt synchronously

Generating a salt via `randomBytes` can also be done synchronously, like so: `var buf = crypto.randomBytes(256);`

The `randomBytes` method will accept a parameter for the size of the generated salt, in bytes. What is returned to us is the randomly generated salt. At this point, we can go to the next step of adding that salt to one our our hash functions, as we'll see in "Choosing the Right Password Hashing Function" on page 35.

Salt Reuse

One of the common issues in using a salt for password hashing is to reuse the salt over and over again. This completely defeats the purpose of using a salt to begin with. If we have a common salt being used, and a series of users that are using the same password, then the resulting hash will be the same. From there, an attacker can create a reverse lookup table and run a dictionary attack on each hash at the same time.

The standard that should be used is that when a user creates a new account, or they change their password, a new salt and hash should be generated and stored.

Salt Length

What is the appropriate length for your salt, and what are the implications of using a salt that is too short?

Let's tackle the first part, what is the ideal length of the salt? One general rule of thumb to use here is for the salt to be the same size as the output of the hash function used. If we look at sha256 for instance, the resulting hash is 32 bytes in length, so our salt should be 32 bytes, at minimum. In the case of sha1, the output length is 20 bytes, so our salt should be 20 bytes as well.

The PBKDF2 standard recommends that a salt length of, at least, 64 bits (8 bytes)[5] be used to be effective. In many cases the next power of 2, so 128 bits (16 bytes), is typically used.

Let's move on to the implications of a using a short salt. Simply put, if the generated salt is short, lookup tables can be created with all possible salt values, and then be used to crack the data.

Where to Store the Salt

The first thought that you may have is that the salt should be stored in a secure location, separate from the hash. The simple fact is that we use the salt to prevent precomputed attacks (e.g. rainbow tables), where we would have a series of hashes that can be compared against what is stored in the user database. If we can prevent that easy / quick lookup from happening, then we force the attacker to start cracking the hashes individually, which is significantly slower.

Since that is the case, we don't need to obfuscate or encrypt the salt, and it can be stored as a plaintext value in our database along with the hash. With that said, you also don't want to make it readily accessible (like a username) to the open world.

Peppering

One of the other concepts in password crytography, beyond the salt, is the concept of a pepper. Much like the salt, a pepper is another value that is added to the salt and password when hasing.

Where the salt and pepper differ is that instead of the pepper being stored alongside the hash (like a public key), and being randomly generated anew for each hash, the pepper instead is a more safeguarded key, and is typically pulled from a single, or subset, of strings.

The simple formula for pepper use is: `hash(salt + pepper + password) = password hash`

When using a pepper, you will generally be following one of two scenarios:

- The pepper value is held in a separate location than the password hash, and is treated more like a private key than a public key.
- The pepper is never stored, but is instead randomly chosen from a subset of values when the hash is generated. When doing a password comparison, you will instead compare the hash to the proposed password, the salt, and each possible

5 *https://tools.ietf.org/html/rfc2898#section-4.1*

value of the pepper, so you will make multiple comparisons against possible values for the pepper until the comparison either passes of fails. This means that the values for the pepper are calculated from the code layer, instead of the stored value.

In general, the reason why one would use a pepper is because the added characters and symbols can be utilized to bolster a weak password. By prepending a unique value that is chosen for a secure approach, we can harden passwords that would otherwise be easily crackable. Your password length is now increased, it has special characters, etc. With this, the resulting hash will be increasingly unique, helping to prevent dictionary attacks.

In reality though, there are a few controversial aspects of a pepper, and are some of the reasons why peppers are not heavily employed:

- A pepper is only valuable if it's kept secret. If an sql injection attack is used, and only one table with the hash and salt are stolen, but the pepper is safe, then it does its job. Unfortunately, in many cases, the entire database structure is compromised and leaked, meaning that our pepper usually goes right along with it.
- Hashing algorithms that are employed en masse, and publically vetted, do not accept a pepper as an argument. Most implementers can bring about some disastrous results by doing this wrong, or modifying the hashing algorithm, which is heavily discouraged.
- There is really no analysis of the benefits of a pepper out there, unlike salting.

Many in the cryptography community simply state that a salt of proper uniqueness and length, and a hash with an appropriate number of iterations employed to slow down cracking, is more than sufficient to make the pepper fairly useless.

In the end, it's good to understand that these mechanisms are out there, but in practicality, it's sometimes more trouble than it's worth, especially when implementing proper hashing and salting.

Choosing the Right Password Hashing Function

Now that we understand how everything works, let's move on to how we are going to pick the hashing function that is best for our needs.

We already know that the three main hashing functions that we should be using for passwords are bcrypt, PBKDF2, and scrypt, but what's the difference between the three? Let's break these down a little bit further and explore the benefits of each, and how they are used in conjunction with a salt.

bcrypt

bcrypt is the first hashing function on our list. It's a key derivation function designed for passwords, and is based on the blowfish cipher.

Some of the benefits of bcrypt are:

- Since it's based on the blowfish cipher, which uses RAM-based lookup tables that are constantly being altered throughout the execution of the algorithm. These types of tables are easy to handle for a CPU, but due to the sequential memory access and parallel processing required, the GPU falls short. In this way, it hinders GPU hardware enhancements by an attacker.
- Specifically designed as a password hashing algorithm, with an intent at being slow (a good thing in password hashing).

With that in mind, let's jump into implementing this into our application or website.

First, we need to install the bcrypt package from npm, like so:

```
npm install bcrypt --save
```

We then require bcrypt in our Node app:

```
var bcrypt = require('bcrypt');
```

The bcrypt packages has a built in method for generating a salt, so we're going to be using that instead of the one that is made available in the crypto library, so that we don't need to include both bcrypt and crypto in our library.

```
function bcrypt_encrypt(username, password){
    bcrypt.genSalt(10, function(err, salt) {
        bcrypt.hash(password, salt, function(err, key) {
            //store username, hashed password, and salt in your database
        });
    });
}
```

We've built a function that accepts a username and password, presumably from user input when they are creating or updating their account. We're taking an asynchronous approach to generating the hash, which is the preferred method. We call `bcrypt.genSalt(…)` to create our salt. The method accepts the number of rounds (or cost) of the process (default is 10), and the callback to the method returns any error, and the derived salt.

Once the salt is generated, we then call the `bcrypt.hash(…)` method to generate our hash. It will accept the password to be hashed, and the salt we just generated. The callback will return the hash key that is generated from the process.

We can then take that hash and store it in our database along with the salt and the rest of the user record.

To do the same thing using a synchronous approach, you can do the following:

```
var salt = bcrypt.genSaltSync(10);
var hash = bcrypt.hashSync(password, salt);
```

PBKDF2

Next, let's look into PBKDF2. A key derivation function that has an academic background, coming from RSA laboratories.

PBKDF2 has a number of main benefits and implementations in the wild:

- Time tested and has been the subject of intense research over the years.
- Recommended by NIST special publication 800-132
- Used by password management systems 1Password, LastPass, and others.
- Available as a standard method within the native Node crypto library.

Since PBKDF2 is the hashing algorithm baked into the Node crypto library, it's fairly easy to get started. We first need to require crypto into our Node application:

```
var crypto = require('crypto');
```

Now, we simply build a function much like we did with bcrypt, to accept a username and password.

```
function pbkdf2_encrypt(username, password){
    crypto.randomBytes(32, function(ex, salt){
        crypto.pbkdf2(password, salt, 4096, 512, 'sha256', function(err, key) {
          if (err) throw err;
          //store username, hashed password, and salt in your database
        });
    });
}
```

We make a request to `crypto.randomBytes(…)` to generate a random salt for us. The method accepts the number of bytes of data to be generated (in our case 32 bytes), and returns a salt.

We then make a request to `crypto.pbkdf2(…)`, passing in:

- The user password
- The salt
- The number of iterations, or the number of times that the hash function should be applied. In our case, 4096.
- The keylength. In our case, 512.

- The digest function. In our case, sha256. You can get a list of supported digest functions with crypto.getHashes().

What is returned to us is a hex string containing our hash. You can push it to a plain string for viewing like so:

```
console.log('key: ' + key.toString('hex'));
```

Like before, you then store your username, hash, and salt in the user database.

To do the same thing synchronously:

```
const salt = crypto.randomBytes(32);
var result = crypto.pbkdf2Sync(password, salt, 4096, 512, 'sha256');
```

scrypt

Last on our hashing function list is scrypt. While there are many heated debates on the use of PBKDF2 vs. bcrypt, and which is better, scrypt takes a very different approach to hashing than either of the two.

Benefits and implementations of scrypt include:

VERIFY THE BELOW - SEEMS TO OVERLAP INCORRECTLY WITH PBKDF2

- Specifically designed to make it hardware and memory intensive for an attacker to perform large scale attacks.
- Implemented as the algorithm behind cryptocurrencies Litecoin and Dogecoin.

The main benefit here is that, unlike bcrypt and PBKDF2, scrypt is designed to be incredibly hardware and memory intensive in order to crack. In the case of bcrypt and PBKDF2, an attacker would be able to run thousands of parallel attacks on the hashed data from minimal hardware resources, since they are not meant to have large resource demands.

Let's get into the implementation. First, let's install scrypt using npm:

```
npm install scrypt --save
```

With scrypt, we're going to use a mix of the crypto library (for the salt), and the scrypt module (for the hash). We include those two into our Node application like this:

```
var scrypt = require('scrypt'),
    crypto = require('crypto');
```

With everything in place, we again have a function that accepts a username and password.

```
function scrypt_encrypt(username, password){
    crypto.randomBytes(32, function(ex, salt){
        scrypt.hash(password, {"N":16384,"r":8,"p":1}, 64, salt, function(err, key) {
            //store username, hashed password, and salt in your database
        });
    });
}
```

We use the crypto library to generate our salt, with `crypto.randomBytes(…)`, passing in the number of bytes that should be generated in the output. The output gives us the generated salt.

We then go to our next step of generating the hash with that salt. We make a request to `scrypt.hash(…)`, which accepts a number of values:

- The user password to be hashed.
- An object containing the parameters to control the scrypt hashing:
  - — N: The maximum amount of time in seconds that scrypt will spend computing the derived key (double).
  - — r: The maximum number of bytes of RAM used when computing the derived key (integer). The default is 0.
  - — p: The fraction of the available RAM used when computing the derived key (0 to 1, converted to percentages). The default is 0.5.
- The length of the resulting hash.
- The salt we just generated.

Once computed, the derived hash will be sent back for us to store.

The same approach synchronously would look like:

```
    const salt = crypto.randomBytes(256);
    var result = scrypt.hashSync(key,{"N":16384,"r":8,"p":1}, 64, salt);
```

Validating a Password Against a Hashed Value

Once we have a hashed valued of a password stored with the salt in our database, how do we validate that the hash we have stored matches a login attempt by a user at future iterations of the application use?

Since we are working with one way hash functions, there is actually a simple way to validate a hash against another password to see if they are valid. We just follow a few steps:

- We capture the password from the user login attempt
- We lookup the record from our database that matches who the user is purporting to be, and get the hash and the salt.

- We use the same hashing function that we did to derive that hash (e.g. bcrypt), with the salt from the database, to generate a new hash.
- We compare the newly generated hash with the hash from the database. If they match, we have a valid password.

If we follow that process for PBKDF2, we can see how comparison will work:

```
var dbsalt = 'USER RECORD SALT FROM YOUR DATABASE';
var dbhash = 'USER RECORD KEY FROM YOUR DATABASE';

crypto.pbkdf2(password, dbsalt, 4096, 512, 'sha256', function(err, comparehash){
    if (err) throw err;
    if (dbhash.toString('hex') === comparehash.toString('hex')){
        //passwords match
    } else {
        //passwords don't match
    }
});
```

As you can see, the code above looks quite familiar to the encryption process with PBKDF2. There are several things that we need to note about the comparison process:

- We first capture the hash and salt from our local user record storage for the user that is trying to login in (variables dbsalt and dbhash).
- We then encrypt the password for the login attempt (the password that the user supplied to log in) and encrypt it using dbsalt, the same salt that encrypted the user password during registration.
- Once that hash is produced (the comparehash variable), we then compare the hash stored for that user in their user record (dbhash) against the newly generated version.
- If the password was the same, using the same salt to encrypt the password and compare against the stored value, then we will have a matching password and can safely log the user in.

Some packages, such as bcrypt, make that process a bit easier for us by providing a compare method. The compare method simplifies the above steps of having to encrypt the password with the salt that we stored by mixing hashing and comparison into a single call, like so:

```
bcrypt.compare(password, hash, function(err, res) {
    //returns true or false
});
```

In this case, we're supplying the raw user password from the login attempt and the hash from the database. The result will be either true or false, depending on if they're a match.

Key Stretching

One of the underlying concepts that makes bcrypt, scrypt, and PBKDF2 effective is a technique that they employ called key stretching. As we learned about in Chapter 1, a vast majority of people don't utilize standards of significant password length and complexity to keep their profile data secure, and prevent attack vectors like brute force attacks, on their own. This is where key streching comes in, it takes a potentially weak password, and outputs an encrypted key of significant length and complexity, that attack vectors like bruce forcing no longer become viable options.

In the case of our cryptographic hash functions, key stretching is done by applying the hash function repeatedly in a loop, until a hash of the desired length and complexity is obtained. When we talked about number of iterations in the hash function examples previously, that is the implementation of this key stretching concept.

Recomputing Hashes

At some point, you may have the need to generate new secure password hashes for your users. Perhaps:

- Hardware has changed due to Moore's Law[6] and you need to change the weight / work factor used by your encryption algorithms.
- Algorithms have changed, and the one you are using is no longer secure, or something better has come along.
- The hashes are simply no longer as secure as they can be.

In these instances, the standard practice is to stored a new hash for the user as they use your system. As each user logs in with their username / password, you log them in as you normally would by comparing the login hash with the stored hash. Then, instead of throwing out the password, you instead generate a new hash for the user, and replace the old one in the user database record, before throwing out the password.

To speed this process along, you can force logout all users. For instance, if you allow users to remain logged in via a session cookie, you can invalidate all user cookies and force each user to log in on their next visit.

6 *https://en.wikipedia.org/wiki/Moore%27s_law*

Next Steps

Expanding upon the concepts of password security, let’s look at a few practical approaches to protecting our systems against attack vectors.

CHAPTER 3

Identity Security Fundamentals

Tim Messerschmidt and Jonathan LeBlanc

After we have taken the time to discuss ongoing issues with current security models in the first chapter and introduced the concepts of secure passwords, hashing and salting in the second chapter, we will now focus on discussing the concept of a person's identity across multiple sites.

> the qualities, beliefs, etc., that make a particular person or group different from others[1]
>
> —Merriam Webster Dictionary

The basic assumption is that differences are what make identity relevant and help using ongoing identity concepts in order to handle authentication and authorization scenarios.

Understanding the concept of various identity types

While using the internet a person establishes a so-called online identity[2] that represents certain elements or characteristics of the human being. This form of identity can - and often will - differ across multiple sites and leads to a fragmentation that we can group into different areas based on a website's use-case.

Based on this we'd like to introduce three different types of identity that we will then discuss in detail: Social Identity, Concrete Identity and Thin Identity. These types of identity often overlap and can share the same attributes.

1 *http://www.merriam-webster.com/dictionary/identity*

2 *http://en.wikipedia.org/wiki/Online_identity*

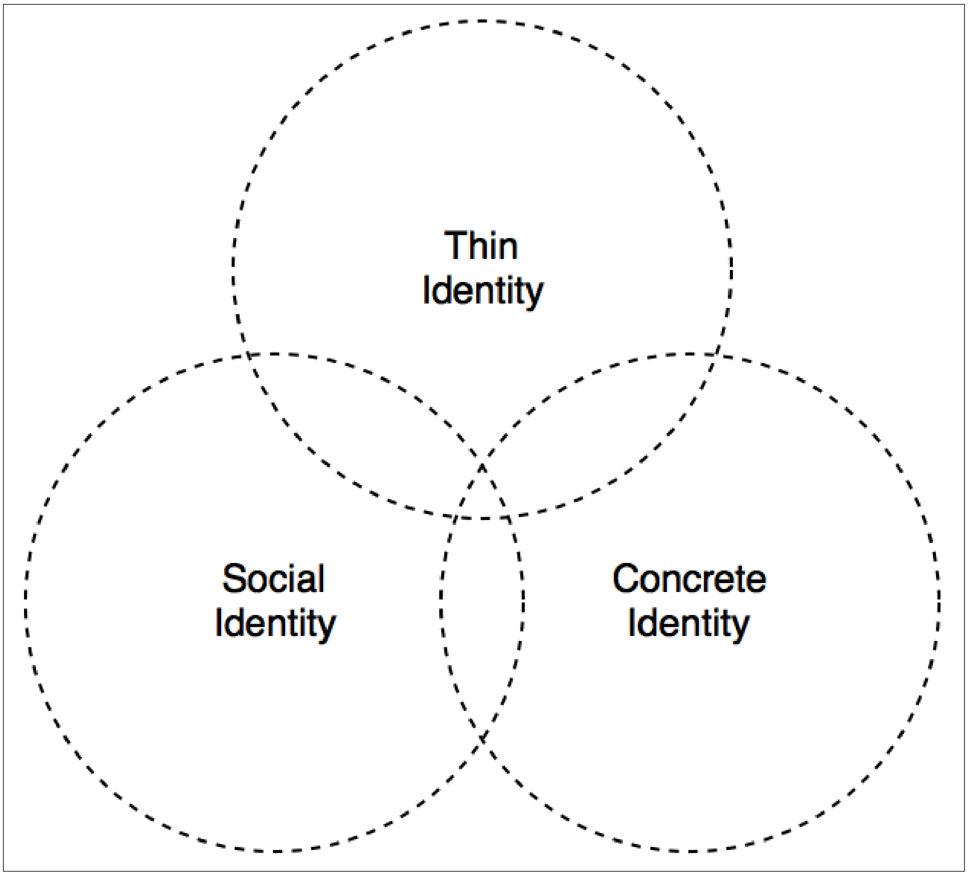


Figure 3-1. Overlapping identities

Federated Identity Management

Often applied through Single sign-on - known as SSO - Federated Identity Management (FIM or FIdM) is the practice of using a set of identity attributes across multiple systems or organizations. The three identity types can be considered federated identities and are applied through technologies such as SAML, OpenID, OAuth and tokenization.

While SSO is a way of using the same credentials across multiple sites and applications, FIM shifts the verification of credentials towards the identity provider.

Social Identity

Social identity came up with the rise of social networks and can be seen as a very moderate form of identity that people tend to share quite casually. Profile information often concentrate on social connections, interests and hobbies, while ignoring or not necessarily favoring severe information that might be used against the user.

Services such as Facebook or Google+ allow the user to quickly access other services using their already populated profiles and leverage scopes in order to control the level of information shared. Especially on mobile phones this quickly became a very much favored way of handling login scenarios - this can be attributed to the issue of entering any kind of complex information on touchscreens and more often than not provides a big boost in convenience.

Concrete Identity

While leveraging social identity is completely valid and even encouraged for services such as games, media consumption and of course social networks, for other use cases such as online banking or eCommerce, a more concrete profile is required that provides useful insights such as the user's email, address, phone number, spoken languages or age.

Especially in eCommerce scenarios the payment process can be painful. Having to enter a 16+ digits credit card number manually can be tedious on a physical device and painful on a touchscreen. This is where services such as PayPal, Amazon Payments or Google Wallet come in. By providing valuable information such as the user's address and tokenizing sensible credentials such as the payment details, the actual checkout flow can be sped up tremendously.

Another popular example of using concrete identity is the election process in Lithuania (and many other state services) are backed up by a state-provided OpenID in the citizen's ID card[3]. This enables a form of eGovernment that enables people living remotely to participate in ongoing discussions and actively contribute to the country's politic environment.

Thin Identity

Thin identity is an old concept that currently gains in popularity again. Thin identity - or even **no** identity - simply means user authentication without actually gaining access to profile information.

A good example would be Twitter's service Digits[4] that allows to use your phone number as a means of logging in. The identifying - and globally unique - bit here is the person's phone number. Looking at the definition of identity that we have introduced in the introduction of this chapter, the criteria of difference (towards other

3 *http://lists.openid.net/pipermail/openid-eu/2009-February/000280.html*

4 *http://get.digits.com/*

phone numbers) is certainly met. Digits - and other similar services - aim at replacing error-prone and vulnerable passwords with another factor that seems to be universally given. Yahoo went a similar route and provided a way to do passwordless login using text messages with one-time only passwords[5] - this is not yet part of Yahoo's developer offerings, though.

Enhancing user experience by utilizing identity

User experience studies carried out by the Nielsen Norman Group[6] show that login doesn't necessarily have to be the first point of contact for users and often harms the conversion process of turning visitors into users by forcing them to register or login. The current sentiment in user experience research is, that a preview of offered functionality is desirable and helps people decide if they want to commit to an application.

Leveraging existing profiles, such as a user's social identity, can help easing the way once the user did decide to register by pre-populating profile information and therefore lowering the amount of information the user has to type in manually.

Introducing the concept of Trust Zones

The devices we use nowadays come pre-equipped with a variety of different sensors that allow to gather information about the user's environment. GPS, WiFi, cameras, accelerometers, gyroscopes, light and many other sensors allow to build up profiles and identify the user accordingly. Combining this concept with the concept of identity, we can not only identify users - we can also build up so-called trust zones.

5 *http://www.infopackets.com/news/9545/new-yahoo-login-system-uses-no-password*

6 *https://www.nngroup.com/articles/login-walls/*

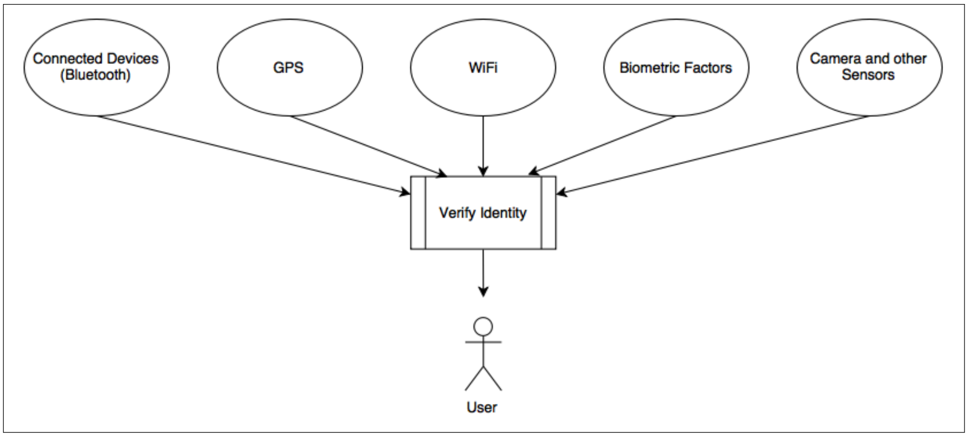


Figure 3-2. The Concept of Trust Zones

Trust zones are a concept that allows us to scale our security based on the user's behavior, environment, and our ability to determine whether a user is who they say they are. In essence, what we are trying to do is create a digital fingerprint for the user, from any data that might be available and unique for the given user, such as their browser configuration, hardware, devices, location, etc.

If we can guarantee a user is at home, based on the current GPS coordinates and the WiFi used to connect to the internet, trust zones can offer the user a way to waive certain steps within authorization and authentication of web and mobile applications. Google introduced this concept for Android as a feature known as Smart Lock[7]: When a user wears his Android Wear device, the phone can be set up to automatically unlock whenever a Bluetooth connection between the wearable device and the user's phone is established[8]. Other supported factors for Smart Lock are the user's location, face recognition and on-body detection - a feature that relies on the device's accelerometer. Chapter 5 covers these alternate ways of user authentication more deeply.

Realistically, what we're trying to do here is to remove hurdles during the application experience for the user by trying to answer the question, if we have obtained enough bits of information about the user from the system and devices that they are using to determine that there are almost certainly who they say they are, and they are in a trusted location, is it necessary to challenge them when changing account information, or provide login details during a checkout process instead of providing a 1-click checkout experience.

7 *http://developers.google.com/identity/smartlock-passwords/android/*

8 *http://support.google.com/nexus/answer/6093922?hl=en*

These are the types of things that we can do when we have a strong certainty that the user is who they are purporting to be.

Let's take this conversation into a practical look into this technology, starting with the browser.

Browser Fingerprinting

One of our main goals as application and web developers is to both make the experience of our users secure, and as conveinient as possible. With the concept of trust zones understood, we can start to see how many of the security measures that we can put in place may occur without burdening the user for more information.

One of the methods that can be employed is through the concept of browser fingerprinting. This process involves using unique characteristics about the browser that the user is using, such as headers, fonts, etc, to be able to determine a second factor of authentication based on the browser that they're using.

Back in May of 2010, the EFF (The Electronic Frontier Foundation)[9] published a report[10] from an experiment that they were running on browser fingerprinting, called Panopticlick[11]. From this study, there were some interesting results that were derived from the browsers that were tested during their first 1 million visits. What they found was that:

- 84% of browsers tested had a unique configuration.
- Among browsers that had Flash or Java installed, 94% were unique.
- Only 1% had fingerprints that were seen more than twice

These are obviously not definitive numbers to determine to a near certainty the browser uniqueness of each user, but the numbers of significant enough to be able to easily use these types of tests as a second factor of authentication for our users. When we are able to find a unique browser configuration, we have a high likelihood (99%) of determining that the browser is unique and attributable to that individual. When using this, coupled with additional techniques that we will explore in later sections, we can predict a high degree of certainty that a user is who they say they are. When we have that determination, along with the login mechanisms that the user themselves has used (such as a username / password), then we are able to maintain a high level of confidence to create our trust zones.

9 *https://www.eff.org/*

10 *https://www.eff.org/deeplinks/2010/05/every-browser-unique-results-fom-panopticlick*

11 *https://panopticlick.eff.org/*

These tests used the concept of *bits of entropy* to determine the fingerprint of a browser. From their subset of tests, they noticed that the distribution of entropy observed on a tested browser is typically around 18.1 bits, This means that if a browser was chosen at random, at best we would expect that only 1 in 286,777 other browsers would share the same fingerprint.

In addition to making things easier for the user through these trust zones, there's another benefit to having this information tracked. If we are processing payments for our users, inevitably there will be some disputes over a payment, and who may have made it. Being able to provide information such as these digital fingerprints during dispute resolution can help to provide favorable results during the process.

Configurations More Resistant to Browser Fingerprinting

In their study, the EFF also noticed that there were certain configurations that made the resistance to browser fingerprinting quite high, meaning that they were harder to generate a fingerprint for. These configurations included:

- Browsers with JavaScript, or plugins disabled.
- Browsers with TorButton installed, which anticipated and defended against the fingerprinting techniques.
- Mobile devices, since the lack of configurability of the mobile browser tends to lead to a more generic fingerprint. These devices do tend to not have good interfaces for controlling cookies, so information may be obtained more easily through that method.
- Corporate desktop machines that are precise clones of one another, and don't allow for degrees of configuration.
- Browsers running in anonymous mode.

Identifiable Browser Information

Through the studies that were performed during the Panopticlick project, they were able to assign different entropy bit levels for different configurations types that can be derived from a browser. These included the following:

Table 3-1. Entropy (bits) for Browser Characteristics

Characteristic	Bits of Entropy
User Agent	10.0
Plugins	15.4
Fonts	13.9
Video	4.83

Characteristic	Bits of Entropy
Supercookies	2.12
HTTP ACCEPT header	6.09
Timezone	3.04
Cookies Enabled	0.353

The browser uniqueness report[12], in addition to providing the characteristics, also provided the means through which those values were obtained:

Table 3-2. How Browser Characteristics Were Obtained

Characteristic	Method
User Agent	This was transmitted via HTTP, and logged by the server. It contains the browser micro-version, OS version, language, toolbar information, and other information on occassion.
Plugins	1The PluginDetect JavaScript library was used to check 8 common plugins. Extra code was also used to estimate the current version of Acrobat Reader. The data was then transmitted via AJAX post.
Fonts	A flash or Java applet was used, and the data was collected via JavaScript and transmitted via AJAX post.
HTTP ACCEPT header	Transmitted by HTTP and logged by the server.
Screen Resolution	JavaScript AJAX post.
Supercookies (partial test)	JavaScript AJAX post.
Timezone	JavaScript AJAX post.
Cookies Enabled	Inferred in HTTP, and logged by the server.

Looking at a breakdown of all characteristics, we have a good idea of how to implement these techniques. For the most part, we're pulling data via JavaScript and logging on our server, and at most (in the case of fonts), we have a flash or Java applet doing the work for us.

Capturing Browser Details

Let's take a look at the methods that we can use to begin capturing some of this information from client-side JavaScript. This will be part of the data that we will need in order to start generating a fingerprint for our users, as they come through.

User Agent

Let's start with a simple one, the user agent. This will provide us with quite a bit of information that we can use for the overall fingerprint.

12 *https://panopticlick.eff.org/static/browser-uniqueness.pdf*

To obtain this string, we can use the data within the navigator object, like so:

```
var agent = navigator.userAgent;
```

From this test, you may see a string returned that would look something like the following:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/48.0.2564.116 Safari/537.36
```

There are some important pieces of information contained in this string that we can use, specifically:

- **Mozilla/5.0**: Mozilla-compatible user agent and version. This is used for historical reasons, and has no real meaning in modern browsers.
- **Intel Mac OS X 10_10_5**: The operating system and version.
- **AppleWebKit/537.36**: Web kit and build.
- **KHTML, like Gecko**: Open source HTML layout engine (KHTML), like Gecko.
- **Chrome/48.0.2564.116**: Browser (Chrome) and version.
- **Safari/537.36**: Based on browser (Safari) and build.

Time Zone

Next, let's capture the time zone using *getTimezoneOffset()*. This function will return the offset, in minutes, from GMT. To obtain the number of hours that the user is offset from GMT, we can simply divide that result by 60, like so:

```
var offset = new Date().getTimezoneOffset() / 60;
```

You may notice something strange about the result here. The hour is correct, but the negative / positive identifier is flipped. For instance, if I am on the east coast of the United States (GMT-5), the result returned is 5, not -5. This is because *getTimezoneOffset()* is calculating GMT's offsite from your time zone, not the other way around. If you wish to have it the other way around, simply multiply by -1, like so:

```
var offset = (new Date().getTimezoneOffset() / 60) * -1;
```

Screen Resolution

The screen resolution can be obtained by using the window object. This will give me the screen resolution of the monitor being used, which can be a fairly static indicator for the browser fingerprint.

We can obtain those results with the following snippets:

```
var width = window.screen.width;
var height = window.screen.height;
```

This will give me the given numeric results for the width and height, such as 2560 (width) and 1440 (height) for a screen resolution of 2560x1440.

Plugins

Browser plugin information can garner quite a bit of detail for the fingerprint, and is obtained via *navigator.plugins*. Let's say we want to capture the name of each plugin installed in the browser, and just display those out for the time being. We can do so with the following code:

```
//get plugin information
var plugins = navigator.plugins;
for (var i = 0; i < plugins.length; i++){
   console.log(plugins[i].name);
}
```

JavaScript Library for Plugin Detection

An alternative method for obtaining additional plugin information from the browser is through the PluginDetect JavaScript library.

The information that may be displayed, depending on the plugins installed in the browser, may look something like the following:

```
Widevine Content Decryption Module
Chrome PDF Viewer
Shockwave Flash
Native Client
```

That information can be added to the custom identifiers for the user's browser.

Location Based Tracking

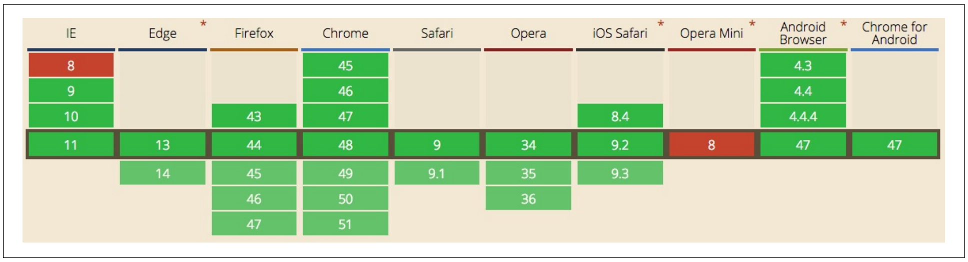
Other than browser fingerprinting, one of the other methods that we can use for building trust zones for a user is to use their physical location.

Here's how this can be valuable. Let's say that we have an ecommerce store where the user has filled out their shipping address during signup, or a previous purchase. We have that address stored, to make it easier for the user the check out, and that has become a trusted home location. If we could determine the physical location of the person that is attempting to use the site while purporting to be the user, we could match that against the address on file. If those two addresses match, we can use that as a trusted point, and potentially lift the need to have the user confirm their login information before checkout, etc.

Use Geolocation with Caution

Use gelocation data from the user with caution. Physical location can be masked, and may provide inaccurate results. With that said, ensure that you use alternate methods of identification with geolocation, and use with caution.

Let's look at a simple JavaScript based approach towards gathering the latitude and longitude of the user, using the navigator object. First off, let's see what the current support for geolocation is within modern browsers:



IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
8			45					4.3	
9			46					4.4	
10		43	47			8.4		4.4.4	
11	13	44	48	9	34	9.2	8	47	47
	14	45	49	9.1	35	9.3			
		46	50		36				
		47	51						

Figure 3-3. Current Geolocation Browser Support

Looking at the support, we have good overall coverage in most modern browsers. Now let's see how to set up a simple example of this.

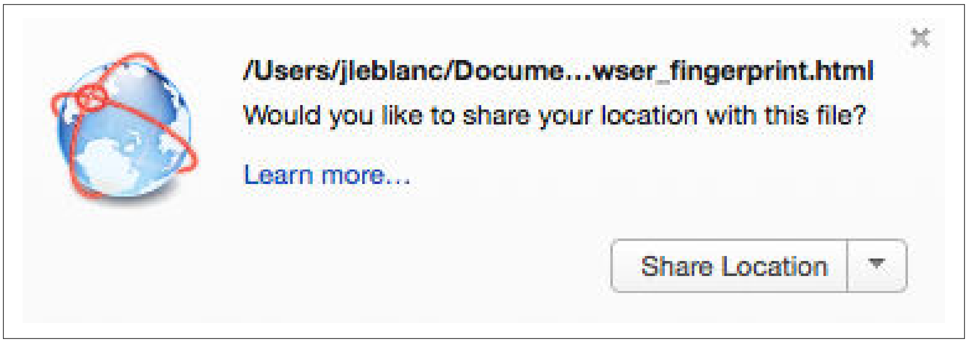
```
//on success handler
function success(position){
    console.log('lat: ' + position.coords.latitude);
    console.log('lon: ' + position.coords.longitude);
}

//error handler
function failure(err){
    console.log(err);
}

//check geolocation browser availability and capture coordinates
if ('geolocation' in navigator){
    navigator.geolocation.getCurrentPosition(success, failure, {timeout:10000});
} else {
    console.log('geolocation is not available');
}
```

We start out by defining two handler functions, one for success and the other for handling errors. Within the success function we will be passed position data, which we can then extract coordinate information from. Within the error handler, we are simply logging out the errors that may be produced. One potential error may be caused by the user not allowing the website to capture their geolocation, which would produce the following error:

```
PositionError {}
 - code: 1
 - message: "User denied Geolocation"
```

With those in place, we check at the bottom of the sample to see if geolocation is available within the navigator object. If it is, we call *navigator.geolocation.getCurrentPosition(…)*, passing in the success function, error function, and the options, which contain a timeout of 10 seconds.

When run in a browser, the user will be asked to confirm the user of their geolocation data.



/Users/jleblanc/Docume…wser_fingerprint.html

Would you like to share your location with this file?

Figure 3-4. Requesting Use of Geolocation Data

Once allowed, we will be able to extract the latitude and longitude, compare to the address we have stored on file, and see if the user is in a trusted zone. Creating a geofence of an appropriate range (range from the root address that the coordinates are within) will allow you to handle cases of the user being within close proximity to their home location.

Other Methods

There are many other methods for obtaining geolocation information for a user. For instance, within a mobile application environment, we can simply leverage off of the GPS data for the same end result.

Device Fingerprinting (Phone / Tablet)

As we're now seeing, there are a multitude of different data points that can be gathered to help us determine if a user is who they say they are, without really impacting their experience. This allows us to continue to make things easier by not having to request additional information from them for security.

Another method that we can use for this cause is to use the hardware fingerprint of the devices that are actually being used by the person using our site or service. Users will typically use a range of devices (phones, tablets, etc) for interacting with your applications. These devices, when used over time, can become trusted sources to help determine if the user is on a trusted device.

Let's take a look at a simple method for capturing this type of information from an Android application. Build[13] information is available to us to be able to obtain information about the device that the user is using.

Some of that information can be pulled like so:

```
//Build Info: http://developer.android.com/reference/android/os/Build.html

System.getProperty("os.version"); //os version
android.os.Build.DEVICE           //device
android.os.Build.MODEL            //model
android.os.Build.VERSION.SDK_INT  //sdk version of the framework
android.os.Build.SERIAL           //hardware serial number, if available
.
.
.
```

We can obtain information such as the OS version, device, model, etc. This can all go towards building a framework of trusted devices, and allowing a user to bypass the need for additional levels of security, should they be required.

Changing Devices

A typical question here may be, "What if I change my device"? If a device is changed, the system should note that the device is not trusted, and show appropriate security challenges as one would for an untrusted user. Once the user has verified their identity through the challenges, that device can then be added to the list of trusted devices.

Device Fingerprinting (Bluetooth Paired Devices)

We are in a time where our phone is not the only connected device we have. We may have our phone connected to a smart watch, a car, or other like hardware around us. These devices, much like the phone, can be used as a hardware fingerprint to help determine if a user is who they say they are. If I can find devices that are typically connected to the phone, then the trust score would increase.

13 *http://developer.android.com/reference/android/os/Build.html*

Let's look at an example of how this would work within an Android application, if I wanted to fetch all of the bluetooth devices that are connected to my phone.

```
//fetch all bonded bluetooth devices
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();

//if devices found, fetch name and MAC address for each
if (pairedDevices.size() > 0){
    for (BluetoothDevice device : pairedDevices){
        //Device Name - device.getName()
        //Device MAC address - device.getAddress()
    }
}
```

We start by calling *getBondedDevices()* to capture any devices that are currently attached to the phone. We then loop through the devices found, and can fetch some basic information about them:

- **Device Name**: Readable name of the device, obtained through *device.getName()*.
- **MAC Address**: The physical address of the device, obtained through *device.getAddress()*.

Setting Proper Permissions

As of Android 6.0, there have been permission changes to provide users with greater data protection. In order to obtain hardware identifiers (such as the MAC address) of a Bluetooth attached device, you need to set the ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permissions in your app. If those permissions are not set, *device.getAddress()* will return a constant value of 02:00:00:00:00:00.[14]

Implementing Identity

Now that we have built up an understanding of different identity types and the concepts behind trust zones, we will take on a basic implementation of OAuth 2.0 and OpenID - the driving technologies behind identity. Please note that the identity sector is currently evolving and new standards - such as FIDO - are on the horizon. These new technologies will be part of Chapter 5's focus.

14 *http://developer.android.com/about/versions/marshmallow/android-6.0-changes.html*

CHAPTER 4

Securing the Login with OAuth 2 and OpenID Connect

Tim Messerschmidt

In this chapter we are going to discuss the concepts behind the two standards *OAuth 2.0* and *OpenID Connect* in order to provide a comprehensive overview of current authentication and authorization standards. In order to do so, the difference between authentication and authorization will be outlined, followed by an explanation of OAuth's evolution throughout the years. Afterwards, we will sketch out a basic implementation of an OAuth 2.0 server and client that leverages OpenID Connect functionality.

The difference between authentication and authorization

A common issue is seeing both authentication and authorization as one and the same. In fact, they are very different and can be used in different scenarios or combined in order to allow for accessing different kinds of information. In order to understand why multiple standards exist and are being pushed forward at the same time, a basic understanding of the main differences will be provided.

Authentication

Authentication is the process of identifying a user against a service. OpenID was the first standard that aimed at providing a decentralized protocol for identifying users across multiple sites. The idea behind this was very simple: avoiding the tedious task of re-entering information over and over. Basically the log-in process is being delegated to another site.

OpenID got introduced in 2005 and saw enormous growth with totaling over 1 billion user accounts in 2009[1]. Recent development showed less demand for OpenID and central identity platforms. Instead hybrid approaches were being introduced that offered both user authentication and authorization at the same time.

Authorization

While authentication aims at user identity, authorization tries to solve the issue of providing access to a user's protected resources. This can involve providing access to user profiles - which blurs the line between authentication and authorization - or simple anomynous access to data.

Authorization standards like OAuth are often used as a more convenient and more secure way of handling sign-in than regular basic authentication flows using usernames and passwords. The idea behind this concept is relying on a third-parties' authentication system. This concept is heavily leveraged for various social login scenarios using Service Providers like Facebook, Google Plus or Twitter.

What is OAuth and OpenID Connect?

The first draft of the OAuth 1.0 Core got released in December 2007[2]. The idea behind OAuth was to provide an authentication technology that would allow for anonymous resource sharing with third-parties. Anonymous resources sharing can be seen as a way of providing access to information and resources without the need of providing information about the user's identity.

The server-side of this process is being titled as Service Provider, the client is also known as Consumer. In order to allow for resource sharing and accessing a user's protected resources, a process called *OAuth authorization flow* is initiated that consists out of 8 steps:

1. Consumer: Retrieve a Request Token
2. Service Provider: Grant Request Token
3. Consumer: Direct user to the Service Provider in order to sign-in
4. Service Provider: Obtain authorization
5. Service Provider: Redirect the user to the Consumer
6. Consumer: Request an Access Token
7. Service Provider: Grant Access Token

1 *http://openid.net/2009/12/16/openid-2009-year-in-review/*

2 *http://oauth.net/core/1.0/*

8. Consumer: Use Access Token to access protected resources

This flow - as outlined in the official specification - is known as three-legged OAuth. 3-legged OAuth requires using a web browser in order to obtain a user's authorization. The three parties involved in the process are:

1. The Consumer (which we also identify as the Client)
2. The Service Provider
3. The End-User

The whole nature and reason for this convoluted process results out of the desire to prevent the Consumer from ever handling the user's credentials (username and password). By involving all three parties only a minimum of information is provided in order to grant access to a user's resources. By implementing OAuth, a potentially insecure password is replaced by an opaque token that can be revoked by the application and the end-user. This results in avoiding the password anti-pattern.

> Users should have access to their data and should be allowed to bring it from one site to another. Social sites shouldn't propagate bad behavior by teaching users that it's ok to give any site their user names and passwords for all the sites to which they belong.[3]
>
> —Designing Social Interfaces

An alternative flow - known as two-legged OAuth - skips obtaining the user's authorization since no user data is being requested or involved. User data can be filled within the Consumer and stored afterwards within the Service Provider. The two-legged OAuth flow can be used as a replacement for traditional basic authentication.

3 *http://designingsocialinterfaces.com/patterns/The_Password_Anti-Pattern*

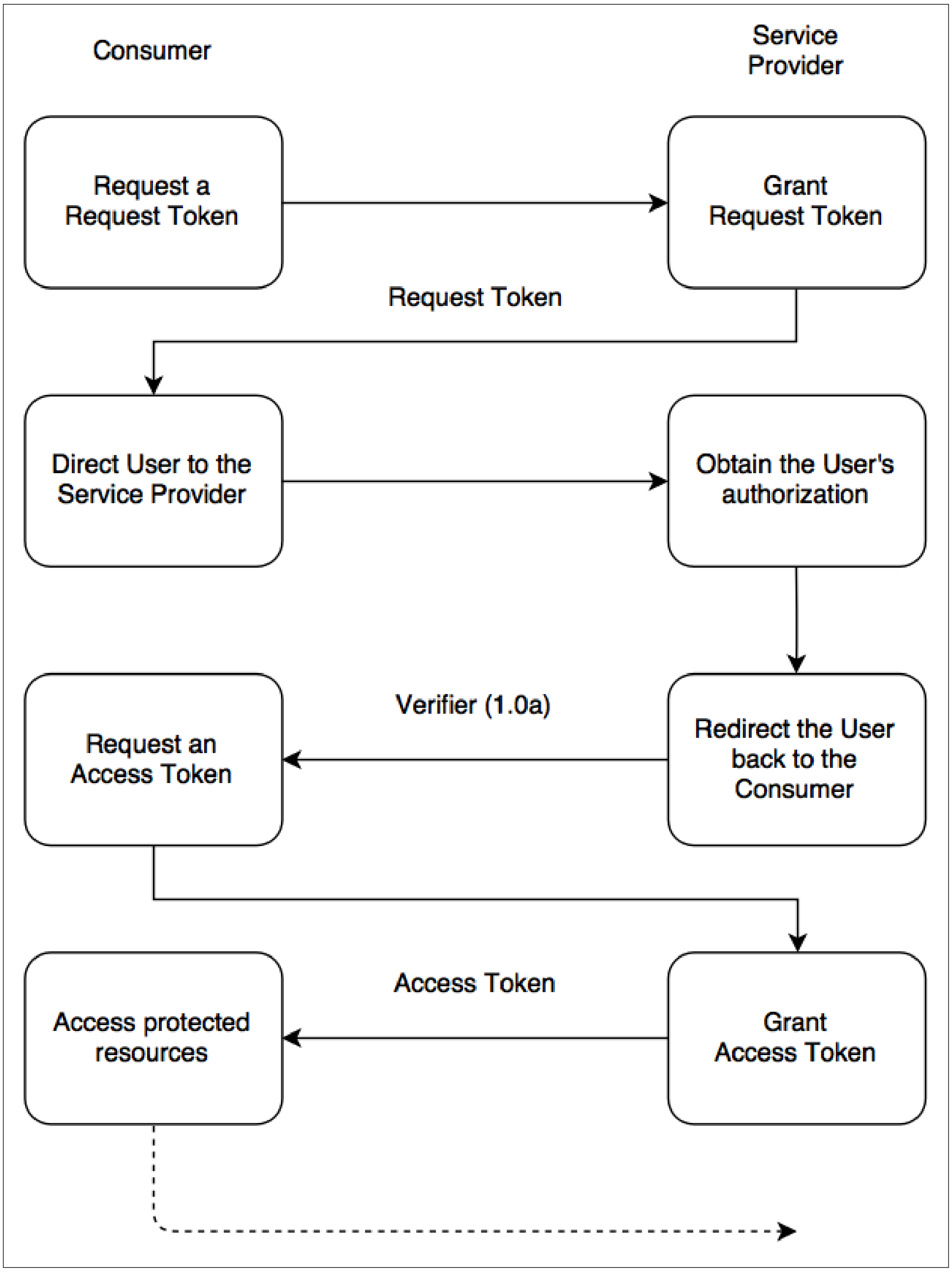


Figure 4-1. OAuth 1.0a authorization flow

The Internet Engineering Task Force - known as IETF - started supporting OAuth in November 2008. The official Request for Comments got published in 2010[4].

OAuth 1.0a

Revision 1.0a of OAuth got released in June 2009. It serves as a hot-fix for possible session fixation attacks that allowed to access the victim's resources.

Eran Hammer, one of the co-authors of OAuth 1.0, wrote a blog post about the possible exploit and how OAuth 1.0a fixes this attack vector[5].

In this book we will use OAuth 1.0a as a base for any discussion regarding the first version of OAuth.

From a security perspective the first version of OAuth heavily relies on signing the request on the Consumer's side using `HMAC-SHA1`, `RSA-SHA1` or `PLAINTEXT`. A Service Provider is allowed to implement other signing methods. The signature is being passed as `oauth_signature` parameter, while the method used to sign the signature is being provided via `oauth_signature_method`. Nonces and timestamps are being leveraged as additional security mechanisms that aim at avoiding replay attacks.

The process of signing requests and the bloated process of retrieving an Access Token can be found amongst the main reasons behind criticism for OAuth 1.0a. Developers often feel like they have to rely on libraries in order to implement OAuth 1.0-based authentication and feel like the standard is not approachable.

Introducing OAuth 2.0

Since the web changed heavily and new form factors got released, new authentication scenarios had to be introduced in order to accommodate web applications, native applications for desktop and mobile, and even interfaceless Consumers such as Internet of Things devices. Furthermore demand for more simplicity has been rising amongst the developer community. Both reasons led to the introduction of the first draft of OAuth 2.0 in April 2010[6]. The main framework for OAuth 2.0 RFC[7] and a RFC discussing OAuth 2.0 Bearer Token Usage[8] have been published in 2012.

4 *http://tools.ietf.org/html/rfc5849*

5 *http://hueniverse.com/2009/04/23/explaining-the-oauth-session-fixation-attack/*

6 *http://tools.ietf.org/html/draft-ietf-oauth-v2-01*

7 *http://tools.ietf.org/html/rfc6749*

8 *http://tools.ietf.org/html/rfc6750*

Amongst the main differences between OAuth 1.0a and 2.0, the following details got updated:

1. Access Tokens are now subject to a time to live (TTL) / expiry time
2. No more client-side cryptography
3. Different flows to accommodate different authentication scenarios

Drama around OAuth 2.0

If you are aware about the ongoing discussion between the different OAuth 2.0 stakeholders or you simply don't care: feel free to skip to the next section!

While OAuth 2.0 fixes a lot of the issues that OAuth 1.0 had, it is far from perfect and saw its biggest critique coming from Eran Hammer, who participated in the OAuth 2.0 working group. In a post called "OAuth 2.0 and the Road to Hell"[9] Hammer writes about his frustration with a standard that often lacks concrete implementation strategies and leaves a lot of decision making to the implementor. One of the points he touches is that Bearer tokens are not encrypted per se and therefore are inherently less secure than specified. OAuth 2.0 puts the trust into TLS and SSL and doesn't add additional security mechanisms on top of these protocols in order to control token ownership.

Other contributors, such as Tim Bray, on the other hand raise valid points about OAuth 2.0 being usable already, working in its core and not necessarily having the need for interoperability.

> Having said all that, OAuth 2 may not be perfect, and may have been harmed by the Enterprise crap, but the core of Web functionality (all I care about) seems to have survived.[10]
>
> —On the Deadness of OAuth 2, *Tim Bray*

OAuth 1.0 vs. OAuth 2.0. Overall the industry seems to agree that OAuth 2.0 is a better standard than its predecessor by offering an easier implementation and more flexibility in terms of defining access to resources. Many developers are heavily challenged when implementing OAuth 1.0's signatures and often run into outdated or even deprecated libraries. When you are able to use secure connections (which we highly rec-

9 *http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell*

10 *https://www.tbray.org/ongoing/When/201x/2012/07/28/Oauth2-dead*

ommend and explain in Chapter 7) it makes sense to go for the slimmer OAuth 2.0 - otherwise we'd recommend to take a deeper look into OAuth 1.0a.

Looking at the tech landscape, there are only few active companies remaining that still build upon OAuth 1.0 (such as Evernote[11]). Twitter, another prominent implementor of OAuth 1.0a nowadays offers a hybrid implementation that uses both OAuth 1.0a and OAuth 2.0[12] based on the API you wish to use. Google, on the other hand, announced that their OAuth 1.0 support has been deprecated as of April 20, 2012 and actively encourages developers to start using OAuth 2.0 instead[13].

Handling authorization with OAuth 2.0

In OAuth 2.0 the classical OAuth authorization flow, also known as *OAuth dance*, was simplified in order to require less steps.

1. Consumer: Direct user to the Service Provider in order to sign-in
2. Service Provider: Obtain authorization
3. Service Provider: Redirect the user to the Consumer
4. Consumer: Use Authorization Code to request Access Token
5. Service Provider: Grant Access Token
6. Consumer: Use Access Token to access protected resources

This flow is a very basic summary of the process that happens when users aim at authorizing clients through OAuth 2.0. Please reference the visualization in order to see a more comprehensive version of the so-called OAuth dance.

11 *http://dev.evernote.com/doc/articles/authentication.php*

12 *http://dev.twitter.com/oauth/reference/post/oauth2/token*

13 *https://developers.google.com/identity/protocols/OAuthForWebApps*

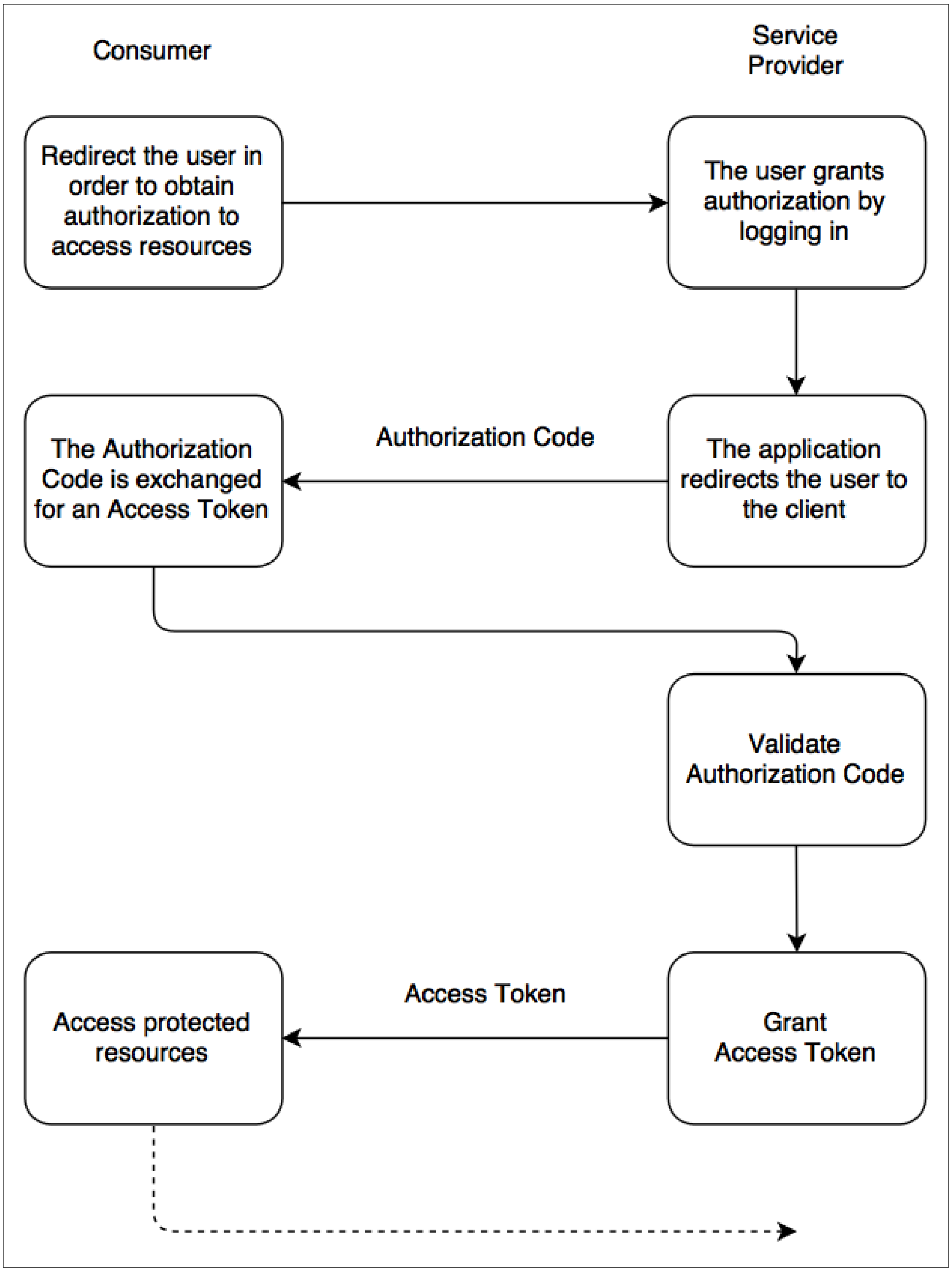

Consumer
Service Provider
Redirect the user in order to obtain authorization to access resources
The user grants authorization by logging in
Authorization Code
The Authorization Code is exchanged for an Access Token
The application redirects the user to the client
Validate Authorization Code
Access Token
Access protected resources
Grant Access Token

Figure 4-2. OAuth 2.0 authorization flow

This flow is also known as Authorization Code flow and will be discussed in detail in this chapter.

OAuth 2.0 Authorization Grant Types

According to the specification, the various OAuth 2.0 Grant Types are defined as:

1. Authorization Code: an intermediate token is being used to prevent sharing the resource owner's credentials
2. Implicit: the client is not being authenticated
3. Resource Owner Password Credentials: used to obtain an authorization grant
4. Client Credentials: when the client is also the resource owner

A less confusing translation is provided by Aaaron Parecki[14]:

- Authorization Code: for apps running on a web server
- Implicit: for browser-based or mobile apps
- Resource Owner Password Credentials: for logging in with a username and password
- Client Credentials: for application access

More information on authorization grant types can be found in Section 1.3 of the OAuth 2.0 RFC document[15].

Using the Bearer Token

The Bearer token is one of the most used default token types amongst the OAuth 2.0 standard. When the server's token endpoint retrieves a request for a new token, it recognizes the type `bearer` and provides a default Access Token according to the specification. This token is not further encrypted or signed - if this is something you are interested in, the token type `MAC` (which stands for Message Authentication Code[16]) is what you're looking for. An alternative to this type is utilizing JSON Web Tokens[17].

14 *https://aaronparecki.com/2012/07/29/2/oauth2-simplified#authorization*

15 *http://tools.ietf.org/html/rfc6749#Section-1.3*

16 *https://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-05*

17 *http://tools.ietf.org/html/rfc7519*

There are three different ways to use Bearer Tokens in practice. The first option is using Request Header Fields to provide the Access Token: `Authorization: Bearer 4ae6ce68-4c59-4313-94e2-fcc2932cf5ca`.

Second, the token can also be passed in the request's body as a form encoded parameter named `access_token`. In order for this to work the request's `Content-Type` header needs to be set to `application/x-www-form-urlencoded` and all further body parameters need to comply to the encoding requirements - JavaScript's method `encodeURIComponent()` comes in handy here.

When providing the Access Token as a URI query parameter, use `access_token` followed by the Access Token itself. This last method is least desirable, since URLs might be logged. If neither using the Authorization header, nor passing the Access Token via the request body is an option for you, make sure to send a `Cache-Control` header that is set to `no-store`. Further security considerations are being outlined in Section 5 of the official Bearer Token RFC[18].

Authorization and authentication with OpenID Connect

Now that we have discussed OAuth 2.0 in detail it is time to highlight OpenID Connect. OpenID traditionally stands for an authentication framework that has been widely adopted in the pre-2010 era. With the rise of OAuth and the users' wish to adopt multiple identities based on the authentication use case, a variety of hybrid extensions and so called pseudo-authentication using OAuth became popular. OpenID Connect is a standard issued by the OpenID Foundation[19] in February 2014[20] and resembles an extra layer on top of the OAuth 2.0 core that handles user authentication in a standardized REST-like manner. All data being transmitted is formatted using JSON. While OAuth is supposed to be a standard for authorization, OpenID Connect enables authentication use cases leveraging the OAuth 2.0 protocol. This pretty much means that OpenID Connect is a superset of OAuth 2.0.

UserInfo Endpoint and Claims

Next to OAuth's Authorization and Token endpoint, OpenID Connect relies on an additional endpoint called *UserInfo*. This endpoint provides *Claims* about the authenticated user. Claims are a pre-defined set of profile information about the user like the user's name, the profile's URL, or if the user's email address has been verified[21]. Additionally an Address Claim can be requested and custom Claims can be defined.

18 *https://tools.ietf.org/html/rfc6750#Section-5*

19 *http://openid.net/foundation/*

20 *http://openid.net/2014/02/26/the-openid-foundation-launches-the-openid-connect-standard/*

21 *http://openid.net/specs/openid-connect-core-1_0.html#StandardClaims*

User Authentication

OpenID Connect allows to handle a user's log in or to determine if a user is logged in already.

The ID Token

A central part of the OpenID Connect specification[22] is the ID Token. The the token's functionality resolves around the following details:

- **serves** as security token
- **contains** authentication information
- **is signed** using JSON Web Signatures (JWS)
- **can be encrypted** using JSON Web Encryption (JWE)

At the time of writing this book, both JWS[23] and JWE[24] are still work in progress.

Security considerations between OAuth 2 and OAuth 1.0a

As highlighted in this chapter's introduction, OAuth 1.0a's security heavily relies on client-side cryptography mechanisms. Using TLS or SSL is not enforced and important credentials like the consumer secret and token secret are stored in plaintext. Phishing attacks are a possibility and require the user to verify the authenticity of websites before providing any credentials[25].

OAuth 2.0 on the other hand relies on TLS version 1.0[26] as a security mechanism. Additionally protection against cross-site request forgery, known as CSRF attacks, can be achieved by utilizing the `state` parameter (as outlined in the Section about Authorization Codes).

22 *http://openid.net/specs/openid-connect-core-1_0.html#IDToken*

23 *https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41*

24 *https://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-40*

25 *http://oauth.net/core/1.0a/#anchor32*

26 *http://tools.ietf.org/html/rfc6749#Section-1.6*

Building an OAuth 2.0 server

In this Section we will discuss an implementation of OAuth 2.0 for Node.js web application framework *Express* [27]. Express serves as base for many web projects and is listed as one of the most popular packages hosted on npm, the Node package manager.

To allow for seamless authentication and authorization mechanisms between the server's different routes, a feature called middleware will be leveraged.

Middleware can be used on application-level, route-level and to handle errors. You can imagine middleware as a function that can be plugged into our application's routes and is able to interact with the request and response objects. Popular examples for third-party middleware are `cookie-parser`, which allows to parse cookie headers, and `passport` which is a popular middleware for handling user authentication.

> Since version 4.x of Express, the only built-in middleware is `express.static`; this function is responsible for serving static assets[28].

Creating the Express application

Before we can get started with the OAuth 2.0 server integration, we need to take a few minutes in order to set up our environment. Please refer to this book's appendix for an instruction on installing and setting up Node.js and the Express generator accordingly.

Following the installation of the generator, we will create our first Express application:

```
express oauth
cd oauth
npm install
npm start
```

This will create a new Express application in the folder `oauth` and install the required dependencies. The command `npm start` runs our application, which will be served from now on `http://localhost:3000/` (unless specified otherwise).

Congratulations - you've just created your first Express application! We will use this knowledge in our upcoming samples throughout this book.

27 *http://expressjs.com/*

28 *http://expressjs.com/en/guide/using-middleware.html#middleware.built-in*

Setting up our server's database

To allow for interaction with MongoDB[29], our project's database, we will use the database connector *mongoose.js* [30]. To install the package execute `npm install mongoose --save` from the root folder of your application. The `--save` option for the `npm install` command makes sure that an entry for mongoose is added to our project's `package.json` file as a run-time dependency. This is important since the `node_modules` folder, where all the installed modules reside, is usually not pushed to the server.

First, we need to set up the code that allows to connect to our MongoDB database. In order to so add the following code snippet to the `app.js` file that the Express generator created for you.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/book');
```

For development purposes we will use a local database - obviously this would change when deploying this application onto a different environment. By connecting to mongoose within `app.js` we provide application-wide access to the database itself - a necessity for having the ability to handle database queries and object manipulation within our application's routes.

Generating Authorization Codes and tokens

There are two types of tokens that are relevant for our OAuth 2.0 integration. A key requirement for OAuth Tokens is that they need to be unique, non-sequential and non-guessable.

Authorization Codes

Authorization codes are used in redirection based authorization flows. A code can only be used once and a maximum lifetime of 10 minutes is recommended by the official specification[31].

A typical authorization server response contains the following parameters:

Table 4-1. Authorization request response

`parameter`	necessity	description
`code`	required	An access grant that is bound to the client's redirect URL and identifier
`state`	required[a]	Used to prevent cross-site request forgery

29 *https://www.mongodb.org/*

30 *http://mongoosejs.com/*

31 *http://tools.ietf.org/html/rfc6749#Section-4.1.2*

parameter	necessity	description

[a] Providing the `state` parameter in the authorization response is required when it got submitted in the authorization request

Access Tokens

Access Tokens are tokens with a short lifetime that are used to access protected resources. The short lifetime is an inbuilt security mechanism that tries to prevent any fraudulent usage of resources. Using the Access Token to access resources consumes the token. Securing HTTP requests to access OAuth 2.0 protected resources is also known as *Bearer Token* usage.

Access Tokens in OAuth 1.0

An Access Token in OAuth 1.0 is valid until it is revoked manually.

Table 4-2. Defining a successful Access Token response

parameter	necessity	description
`access_token`	required	The Access Token issued by the authorization server
`token_type`	required	Helps the client utilize the Access Token with resource requests
`expires_in`	recommended	Lifetime in seconds
`scope`	optional	The scope of resources that can be accessed
`refresh_token`	optional	Used to refresh the Access Token with the same authorization grant

Refresh Tokens

Refresh Tokens allow refreshing Access Tokens after they got consumed or expired. In order to allow for refreshing the Access Tokens, Refresh Tokens are long-lived and expire after they have been used to refresh the Access Token. This feature can also be used to request Access Tokens with a narrower scope and its implementation is optional.

Generation of codes and tokens

To ensure complete uniqueness of tokens and authorization codes, various packages for Node.js can be leveraged in order to generate UUIDs. Amongst the popular modules to generate UUIDs are *hat* [32] and *node-uuid* [33].

32 *http://github.com/substack/node-hat*

33 *http://github.com/broofa/node-uuid*

In order to use hat we will use npm:

```
npm install hat --save
```

After the module is installed and the entry added to our *package.json* file, we can start working with hat in order to create UUIDs:

```
var hat = require('hat');
var token = hat();
```

In order to avoid collision the function rack() can be used:

```
var hat = require('hat');
var rack = hat.rack();
var token = rack();
```

node-uuid, another solution to generate UUIDs, can be installed using npm, too:

```
npm install node-uuid --save
```

Next, we obtain access to the module and generating UUIDs using the *v4* method.

```
var uuid = require('node-uuid');
var token = uuid.v4(); // Unique token
```

Both hat and node-uuid allow for passing additional parameters to further randomize the generated tokens. Please refer to the individual documentations for more details. For any further examples in this book we will stick with using node-uuid. Please note that both modules can be exchanged mutually based on your own preference.

If we were to generate a unique string using a self-implemented method, one might consider JavaScript's Math.random() as a feasible choice to build upon. Considering its pseudo random nature (see the documentation[34]) a different method like the two packages outlined above should be used, though. Adam Hyland published a fantastic article on the nature of Math.random() in 2013 - it should definitely be read when considering the use of this method[35].

Official documentation of Math.random() from ES5

Returns a Number value with positive sign, greater than or equal to 0 but less than 1, chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

34 *http://es5.github.io/x15.8.html#x15.8.2.14*

35 *https://bocoup.com/weblog/random-numbers/*

The correct implementation if UUIDs requires to meet three criteria:

1. universally unique
2. non-sequential
3. non-guessable

Using a database's identifier like MongoDB's `ObjectId` is not recommended since we cannot guarantee the requirements above. `ObjectId` is a 12-byte BSON type and consists of the following elements (according to the documentation[36]):

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

The Authorization Endpoint

As discussed in the introductory section for OAuth 2.0, the Authorization Code flow requires two endpoints to be implemented in order to work. First, we will have a look at implementing the authorization endpoint:

```
var uuid = require('node-uuid');
var Client = require('../lib/models/client');
var AuthCode = require('../lib/models/authcode');

router.get('/authorize', function(req, res, next) {
  var responseType = req.query.response_type;
  var clientId = req.query.client_id;
  var redirectUri = req.query.redirect_uri;
  var scope = req.query.scope;
  var state = req.query.state;

  if (!responseType) {
    // cancel the request - we miss the response type
  }

  if (responseType !== 'code') {
    // notify the user about an unsupported response type
  }

  if (!clientId) {
    // cancel the request - client id is missing
  }
```

36 *http://docs.mongodb.org/manual/reference/object-id/*

```
    Client.findOne({
      clientId: clientId
    }, function (err, client) {
      if (err) {
        // handle the error by passing it to the middleware
        next(err);
      }

      if (!client) {
        // cancel the request - the client does not exist
      }

      if (redirectUri !== client.redirectUri) {
        // cancel the request
      }

      if (scope !== client.scope) {
        // handle the scope
      }

      var authCode = new AuthCode({
        clientId: clientId,
        userId: client.userId,
        redirectUri: redirectUri
      });
      authCode.save();

      var response = {
        state: state,
        code: authCode.code
      };

      if (redirectUri) {
        var redirect = redirectUri +
          '?code=' + response.code +
          (state === undefined ? '' : '&state=' + state);
        res.redirect(redirect);
      } else {
        res.json(response);
      }
    });
  });
```

The code above assumes that there is a mongoose model called `Client`. A client consists of an id, a secret, a user id and a few other attributes like the redirect URI it uses in order to communicate with the Consumer.

When the redirect client flow is being used, the code is provided as query parameter - in case of a resource request, a JSON object containing both the state and the code is returned.

Amongst the attributes of a client is also the so-called scope. The scope tells the Service Provider which kind of attributes the Consumer is allowed to access. When obtaining the user's authorization, clients usually display the scopes in order to make sure that users understand what kind of information they share.

Our application requires client credentials to be passed in both the `/authorization` and `/token` route. Client ids, secrets and names are required to be unique in order to avoid issues - we will build upon MongoDB's schema mechanisms in order to realize this challenge: Properties can be flagged as `unique` in order to avoid duplicate keys in the database.

```
var mongoose = require('mongoose');
var uuid = require('node-uuid');

var ClientModel = function() {
  var clientSchema = mongoose.Schema({
    clientId:     { type: String, default: uuid.v4(), unique: true },
    clientSecret: { type: String, default: uuid.v4(), unique: true },
    createdAt:    { type: Date,   default: Date.now },
    name:         { type: String, unique: true },
    scope:        { type: String },
    userId:       { type: String },
    redirectUri:  { type: String }
  });

  return mongoose.model('Client', clientSchema);
};

module.exports = new ClientModel();
```

In `ClientModel` we use MongoDB's default initialization to our advantage. Instead of having to pass a client id and secret whenever we create a client, this process is shifted towards the database schema itself.

When dealing with attributes that are flagged as `unique` - such as the client's name - we need to check if the database entry was created successfully. When using the `save()` method you'll notice that `mongoose` and the underlying MongoDB won't provide feedback if the operation was successful or not. This is where a callback mechanism can be used. By checking if an error occurred before rendering the client's details, we can ensure that we avoid confusion and problems:

```
router.get('/', function(req, res, next) {
  var client = new Client({
    name: 'Test',
    userId: 1,
    redirectUri: 'http://localhost:5000/callback'
  });
  client.save(function(err) {
    if (err) {
      next(new Error('Client name exists already'));
    } else {
      res.json(client);
    }
  });
});
```

When implementing this route you'll want to pair it with a form that allows to enter a client name, select scopes - which will be required for OpenID Connect - and enter the client's redirect URI.

`AuthCode`, another mongoose model we rely on, is implemented in a similar manner. It represents the authorization codes our application issues in `/authorize`:

```
var mongoose = require('mongoose');
var uuid = require('node-uuid');

var AuthCodeModel = function() {
  var authCodeSchema = mongoose.Schema({
    code:          { type: String,   default: uuid.v4() },
    createdAt:     { type: Date,     default: Date.now, expires: '10m' },
    consumed:      { type: Boolean,  default: false },
    clientId:      { type: String },
    userId:        { type: String },
    redirectUri:   { type: String }
  });

  return mongoose.model('AuthCode', authCodeSchema);
};

module.exports = new AuthCodeModel();
```

Handling a token's lifetime

In this example we will have a look at creating and storing tokens using mongoose.js, handling the token's lifetime and consuming the token afterwards. For our application we will use a Token TTL of 3 minutes.

First we will set up a new mongoose Schema[37] called `Token`. The schema will consist of the details outlined in the OAuth 2.0 specification:

```
var mongoose = require('mongoose');
var uuid = require('node-uuid');

var TokenModel = function() {
  var tokenSchema = mongoose.Schema({
    userId:        { type: String },
    refreshToken: { type: String,   unique: true },
    accessToken:  { type: String,   default: uuid.v4() },
```

37 *http://mongoosejs.com/docs/guide.html*

```
    expiresIn:    { type: String,   default: '10800' },
    tokenType:    { type: String,   default: 'bearer' },
    consumed:     { type: Boolean,  default: false },
    createdAt:    { type: Date,     default: Date.now,  expires: '3m' }
  });

  return mongoose.model('Token', tokenSchema);
};

module.exports = new TokenModel();
```

You will notice that an `expires` flag has been defined. It's set to three minutes and will cause the database entry to be deleted - other values like `1h` or simple integers for seconds can be used here, too. In order to make creating Access Tokens as easy as writing a few lines of code, sensible default values for fields like `tokenType` are being used.

The Access Token is initialized using the node-uuid module in order to populate the `accessToken` and `refreshToken` field. `userId` identifies the resource owner and can be used to consume all Access Tokens that was assigned to the user.

By providing the user's id to both the `RefreshToken` and `Token` objects, we can ensure that we are able to consume all issued tokens at once.

```
var mongoose = require('mongoose');
var uuid = require('node-uuid');

var RefreshTokenModel = function() {
  var refreshTokenSchema = mongoose.Schema({
    userId:     { type: String },
    token:      { type: String,   default: uuid.v4() },
    createdAt:  { type: Date,     default: Date.now },
    consumed:   { type: Boolean,  default: false }
  });

  return mongoose.model('RefreshToken', refreshTokenSchema);
};

module.exports = new RefreshTokenModel();
```

After defining the Access Token and Refresh Token schema, we're able to generate both of them like this:

```
var uuid = require('node-uuid');
var Token = require('../lib/models/token');
var RefreshToken = require('../lib/models/refreshtoken');

var userId = 1; // some id

var refreshToken = new RefreshToken({
  userId: userId
});
refreshToken.save();
```

```
var token = new Token({
  refreshToken: refreshToken.token,
  userId: userId
});
token.save();
```

If we put all of the above together, we can begin with implementing our token endpoint. For the first implementation of this route, we will refrain from handling Refresh Tokens that got obtained before - we will cover handling them in another section a bit further down this chapter after covering some more basics.

```
var AuthCode = require('../lib/models/authcode');
var Client = require('../lib/models/client');
var Token = require('../lib/models/token');
var RefreshToken = require('../lib/models/refreshtoken');

router.post('/token', function (req, res) {
  var grantType = req.body.grant_type;
  var authCode = req.body.code;
  var redirectUri = req.body.redirect_uri;
  var clientId = req.body.client_id;

  if (!grantType) {
    // no grant type passed - cancel this request
  }

  if (grantType === 'authorization_code') {
    AuthCode.findOne({
      code: authCode
    }, function(err, code) {
      if (err) {
        // handle the error
      }

      if (!code) {
        // no valid authorization code provided - cancel
      }

      if (code.consumed) {
        // the code got consumed already - cancel
      }

      code.consumed = true;
      code.save();

      if (code.redirectUri !== redirectUri) {
        // cancel the request
      }

      // validate the client id - an extra security measure
      Client.findOne({
        clientId: clientId
      }, function(error, client) {
        if (error) {
          // the client id provided was a mismatch or does not exist
        }

        if (!client) {
          // the client id provided was a mismatch or does not exist
        }

        var _refreshToken = new RefreshToken({
          userId: code.userId
        });
        _refreshToken.save();

        var _token = new Token({
          refreshToken: _refreshToken.token,
          userId: code.userId
        });
        _token.save();

        // send the new token to the consumer
        var response = {
          access_token: _token.accessToken,
          refresh_token: _token.refreshToken,
          expires_in: _token.expiresIn,
          token_type: _token.tokenType
        };

        res.json(response);
      });
    });
  }
});
```

Now our server is ready to issue Access Tokens in order to allow the Consumer to access protected resources.

Handling Resource Requests

Whenever a resource is accessed using the Access Token, the token needs to be consumed in order to make sure no more resource requests are made using this token.

```
var Token = require('../lib/models/token');
var accessToken = 'some uuid';

Token.findOne({
  accessToken: accessToken
}, function(err, token) {
  if (err) {
    // handle the error
  }
```

```
    if (!token) {
      // no token found - cancel
    }

    if (token.consumed) {
      // the token got consumed already - cancel
    }

    // consume all tokens - including the one used
    Token.update({
      userId: token.userId,
      consumed: false
    }, {
      $set: { consumed: true }
    });
  });
```

Mongoose's `findOne` and `update` functions are very practical when dealing with tokens since we can easily consume all tokens for a certain user or check if a token is still valid.

This method makes for a very convenient middleware that protects our applications resources. Let's apply this to a Express route:

```
var Token = require('../models/token');

var authorize = function(req, res, next) {
  var accessToken;

  // check the authorization header
  if (req.headers.authorization) {
    // validate the authorization header
    var parts = req.headers.authorization.split(' ');

    if (parts.length < 2) {
      // no access token got provided - cancel
      res.set('WWW-Authenticate', 'Bearer');
      res.sendStatus('401');
      return;
    }

    accessToken = parts[1];
  } else {
    // access token URI query parameter or entity body
    accessToken = req.query.access_token || req.body.access_token;
  }

  if (!accessToken) {
    // no access token got provided - cancel with a 401
  }
```

```
    Token.findOne({
      accessToken: accessToken
    }, function(err, token) {
      // Same as in above example
      ...

      // consume all tokens - including the one used
      Token.update({
        userId: token.userId,
        consumed: false
      }, {
        $set: { consumed: true }
      });

      // ready to access protected resources
      next();
    });
  };

  module.exports = authorize;
```

After the authorization middleware has processed the request, the request is passed on to the next middleware or the route itself by executing `next()`.

Authorizing requests using the authorization middleware we just implemented is as easy as adding it to our resources route:

```
var express = require('express');
var router = express.Router();
var authorize = require('../lib/middleware/authorize');

router.get('/user', authorize, function (req, res) {
  var user = {
    name: 'Tim Messerschmidt',
    country: 'Germany'
  }
  res.json(user);
});

module.exports = router;
```

Additional middleware can be passed by using array-syntax instead. If we were to use another middleware that logs all requests, the `/user` definition would change to the following:

```
router.get('/user', [logger, authorize], function (req, res) {
  var user = {
    name: 'Tim Messerschmidt',
    country: 'Germany'
  }
  res.json(user);
});
```

Using Refresh Tokens

The Refresh Token is being used in order to obtain a new Access Token. In order to do so the Consumer communicates with the Service Provider's token endpoint. In the next example we will continue working on the token endpoint we have implemented before in order to issue Access Tokens in exchange for Authorization Codes.

The key difference for this scenario is the different Grant Type `refresh_token`; it indicates that the client obtained an Access Token before and is now trying to obtain new credentials in order to continue accessing protected resources.

```
var AuthCode = require('../lib/models/authcode');
var Token = require('../lib/models/token');
var RefreshToken = require('../lib/models/refreshtoken');

router.post('/token', function(req, res) {
  var grantType = req.body.grant_type;
  var refreshToken = req.body.refresh_token;
  var authCode = req.body.code;
  var redirectUri = req.body.redirect_uri;
  var clientId = req.body.client_id;

  if (!grantType) {
    // no grant type provided - cancel
  }

  if (grantType === 'authorization_code') {
    ...
  } else if (grantType === 'refresh_token') {
    if (!refreshToken) {
      // no refresh token provided - cancel
    }

    RefreshToken.findOne({
      token: refreshToken
    }, function (err, token) {
      if (err) {
        // handle the error
      }

      if (!token) {
        // no refresh token found
      }

      if (token.consumed) {
        // the token got consumed already
      }

      // consume all previous refresh tokens
      RefreshToken.update({
        userId: token.userId,
        consumed: false
      }, {
        $set: {consumed: true}
      });

      var _refreshToken = new RefreshToken({
        userId: token.userId
      });
      _refreshToken.save();

      var _token = new Token({
        refreshToken: _refreshToken.token,
        userId: token.userId
      });
      _token.save();

      var response = {
        access_token: _token.accessToken,
        refresh_token: _token.refreshToken,
        expires_in: _token.expiresIn,
        token_type: _token.tokenType
      };

      // send the new token to the consumer
      res.json(response);
    });
  }
});
```

You will notice that using the token endpoint with Refresh Tokens is very similar to the code we have used before to authorize requests using Access Tokens. After applying some basic parameter checks, the tokens are being consumed using the `update`-mechanism and the next action - in this case issuing a new Access Token - is executed.

Handling errors

In this section's code listings, we've mostly looked at success cases and covered error-handling mostly through comments. This subsection will cover how to handle errors according to the OAuth spec utilizing Express's `response` object.

Before we dive into OAuth's specification again, a quick look into HTTP status[38] codes reveals, that there is a selection of codes that will be interesting for us:

38 *https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*

code	name	description
`302`	Found	The user agent MUST NOT automatically redirect. Can be used in routes such as the Authorization Request.
`400`	Bad Request	Malformed request.
`401`	Unauthorized	Authentication failed or not provided. Response must contain `WWW-Authenticate` header.
`403`	Forbidden	Even though an authentication might have happened, the requesting party is not authorized to access the underlying resources.
`500`	Internal Server Error	An unexpected condition causes that the server cannot handle the request.
`503`	Service Unavailable	The server might be overloaded and cannot handle the request at this time.

These status codes provide us with a toolkit to handle different use-cases in the authentication and authorization we encounter throughout the OAuth flow. OAuth's specification provides error codes[39] that need to be provided in order to help the implementor with identifying potential sources of mistaking.

code	description
`invalid_request`	Parameters missing, invalid parameter value provided or parameters might be duplicate.
`unauthorized_client`	Malformed request.
`access_denied`	The resource owner or authorization server denied the request.
`unsupported_response_type`	The authorization server does not support obtaining an authorization code using this method.
`invalid_scope`	The requested scope is invalid, unknown, or malformed.
`invalid_grant`	The provided authorization grant or Refresh Token is invalid, expired or the client details don't match those defined in the authorization request (redirect URI, different client).
`server_error`	Internal server error that can be used when a 500 error cannot be returned to the client.
`temporarily_unavailable`	The server is currently unable to handle the request. Can be used in redirect scenarios where a 503 cannot be returned.

Please note that these error codes differ based on the current step in the authentication flow.

Using the knowledge we've acquired about the tools that both HTTP/1.1 and OAuth 2.0 provide, we can advance to building our own error handling class:

```
var util = require('util');

function OAuthError(code, message, err) {
  Error.call(this);
```

39 *http://tools.ietf.org/html/rfc6749#section-4.1.1*

```
    Error.captureStackTrace(this, this.constructor);

    if (err instanceof Error) {
      this.stack = err.stack;
      this.message = message || err.message;
    } else {
      this.message = message || '';
    }
    this.code = code;

    switch (code) {
      case 'unsupported_grant_type':
        this.status = 400;
        break;
      case 'invalid_grant':
        this.status = 400;
        break;
      case 'invalid_request':
        this.status = 400;
        break;
      case 'invalid_client':
        this.status = 401;
        break;
      case 'invalid_token':
        this.status = 401;
        break;
      case 'server_error':
        this.status = 503;
        break;
      default:
        // Leave all other errors to the default error handler
        this.status = 500;
        break;
    }

    return this;
  }

  util.inherits(OAuthError, Error);

  module.exports = OAuthError;
```

Since `OAuthError` is an extension of Error and is supposed to be provided towards Express's error-handling mechanism (via using `next` or throwing an error), Node's `util` module is used in order to inherit all Error methods and properties. The OAuth specification allows to define custom error codes (such as `invalid_token`) in order to refine the interaction with clients.

Since the routes handling `POST` requests don't have access to the `next` parameter, we set up new handler:

```
function handleError(err, res) {
  res.set('Cache-Control', 'no-store');
  res.set('Pragma', 'no-cache');

  if (err.code === 'invalid_client') {
    var header = 'Bearer realm="book", error="invalid_token",' +
      'error_description="No access token provided"';
    res.set('WWW-Authenticate', header);
  }
  res.status(err.status).send({
    error: err.code,
    description: err.message
  });
}

module.exports = handleError;
```

The default behavior is to turn off caching by providing both the `Cache-Control` and `Pragma` (nowadays mostly obsolete) headers to the client. This ensures the freshness of information provided.

Using Error Handlers

After we've extended the Error class in order to be able to provide more meaningful feedback to the client, we are able to implement this accordingly in order to handle all cases that apply to our current scenarios:

The first step is to make both the `OAuthError` class and the `handleError` function available via `require`.

```
// Require custom error handling
var OAuthError = require('../lib/errors/oautherror');
var errorHandler = require('../lib/errors/handler');
```

For `GET` requests we can build upon middlewares once more. By using the command-chain we simply hand over the `OAuthError` to the appropriate handler - the Express generator creates one by default that renders the status code and error message (when running in a development environment):

```
// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res) {
    console.log('error');
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}
```

```
// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res) {
  console.log('error');
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

When checking if the `GET` request was populated according to OAuth's requirements, we can simply invoke the error handler like this:

```
if (!responseType) {
  next(new OAuthError('invalid_request',
    'Missing parameter: response_type'));
}
```

For `POST` requests we alter the code slightly. In this example we validate if the Authorization Code has been consumed before issueing a new Access Token:

```
if (code.consumed) {
  return errorHandler(new OAuthError('invalid_grant',
    'Authorization Code expired'), res);
}
```

You will notice that this time we invoke the `errorHandler` and pass the response object in order to set the status and error message accordingly.

Adding OpenID Connect functionality to the server

After we've discussed how to implement OAuth 2.0 on the server by adding both the `/token` and `/authorize` endpoint, in this section we will highlight how to build upon our existing implementation by adding OpenID Connect functionality on top. Before we add the `userinfo` endpoint, we should examine the ID Token, OpenID Connect's security mechanism of proving authentication and authorization.

Table 4-3. The basics of an ID Token[a]

`parameter`	necessity	description
`iss`	required	Issuer Identifier - comes as URL using `https` scheme and contains host, port, path `https://example.com`
`sub`	required	Subject Identifier - MUST NOT exceed 255 ASCII characters
`aud`	required	The ID Token's audience - MUST contain the OAuth `client_id`
`exp`	required	Expiration time
`iat`	required	Time at which the token was issued

parameter	necessity	description
nonce	required*	If present the nonce must be verified
auth_time	optional	Time at which the user authentication occurred
acr	optional	Authentication Context Class Reference
amr	optional	Authentication Methods References
azp	optional	Authorized party - MUST contain the OAuth client_id

[a] *http://openid.net/specs/openid-connect-core-1_0.html#IDToken*

OpenID Connect tokens are JSON Web Tokens that must be signed using JWS and can then be optionally signed and encrypted using JWS and JWE. Amongst popular module choices to handle JWT with Node is `njwt`, which handles signing tokens via `HS256` per default[40].

The ID Token Schema

As with the other tokens that we've been using in this example so far, we will create a new Mongoose schema, that will handle the generation and expiry of our ID Token.

```
var mongoose = require('mongoose');
var uuid = require('node-uuid');

var IdTokenModel = function() {
  var idTokenSchema = mongoose.Schema({
    createdAt:  { type: Date,   default: Date.now, expires: '1m' },
    iat:        { type: String, default: Math.floor(new Date() / 1000) },
    exp:        { type: String, default: Math.floor(new Date() / 1000) + 180 },
    sub:        { type: String, default: uuid.v4(), maxlength: 255 },
    iss:        { type: String },
    aud:        { type: String },
    userId:     { type: String }
  });

  return mongoose.model('IdToken', idTokenSchema);
};

module.exports = new IdTokenModel();
```

The ID Token specification[41] requires the `iat` and `exp` values to represent the seconds since 1970-01-01T0:0:0Z UTC. We achieve this by simply using JavaScript's Date class[42] which returns the milliseconds since Unix Epoch[43] and get the seconds by

40 *https://www.npmjs.com/package/njwt*

41 *http://openid.net/specs/openid-connect-core-1_0.html#IDToken*

42 *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date*

43 *https://en.wikipedia.org/wiki/Unix_time*

dividing the result through 1000. `Math.floor` rounds down the result to the next integer.

Modifying the Authorization Endpoint

When dealing with a regular OAuth 2 authorization flow, the scope parameter is considered optional and serves the purpose of defining resources that will be accesses on top of handling a simple login with OAuth. This behavior changes when we decide to use OpenID Connect as our Authorization and Authentication framework, though. OpenID Connect's specification defines multiple scope values that can be passed in order to specify which profile information are required by the client. The minimum scope that needs to be passed is `openid`; this tells the server that an OpenID Connect authorization attempt is being made.

Considering the required scope as outlined above we will go ahead and modify the authorization endpoint accordingly:

```
router.get('/authorize', function(req, res, next) {
  var responseType = req.query.response_type;
  var clientId = req.query.client_id;
  var redirectUri = req.query.redirect_uri;
  var scope = req.query.scope;
  var state = req.query.state;
  var userId = req.query.user_id;

  // Same as in above example
  ...

  if (!scope || scope.indexOf('openid') < 0) {
      next(new OAuthError('invalid_scope',
        'Scope is missing or not well-defined'));
  }

  Client.findOne({
    clientId: clientId
  }, function (err, client) {
    ...

    if (scope !== client.scope) {
      next(new OAuthError('invalid_scope',
        'Scope is missing or not well-defined'));
    }

    ...
  });
});
```

In the above example we have made the basic assumption that each request to the endpoint is an OpenID Connect authentication request. Another way of handling the request is handling all requests with a scope containing `openid` as OpenID Connect and all other requests as OAuth 2.

Instead of checking the scope parameter in the initial statements, we move the conditional statement down to the Client section and will check if we deal with an OpenID Connect authentication request. The OpenID Connect specification does not specify how to handle non `openid` scope requests - the fallback to OAuth 2 seems to be a sensible choice, though.

```
if (scope && scope.indexOf('openid') >= 0) {
  // OpenID Connect Authentication request - generate an ID Token
}
```

Adjusting the Token Endpoint

While the changes to the Authorization Endpoint are very minimal and easy to handle, we need to do a few more changes to our Token Endpoint. The client's request will practically stay the same but we'll need to create the ID Token, store that token and pass it to the client with our Access Token response.

One of the new requirements for the Token Endpoint is verifying if the Grant Type (the Authorization Code we pass to the Token Endpoint) is the result of an OpenID Connect authentication request - otherwise the endpoint is not supposed to return an ID Token.

```
router.post('/token', function (req, res) {
  var grantType = req.body.grant_type;
  var refreshToken = req.body.refresh_token;
  var authCode = req.body.code;
  var redirectUri = req.body.redirect_uri;
  var clientId = req.body.client_id;

  if (!grantType) {
    return errorHandler(
      new OAuthError('invalid_request',
      'Missing parameter: grant_type'),
      res);
  }

  if (grantType === 'authorization_code') {
    AuthCode.findOne({
      code: authCode
    }, function (err, code) {
      // the same validation as for the OAuth 2 flow
      ...

      Client.findOne({
        clientId: clientId
      }, function (error, client) {
        // same as in the OAuth 2 example
        ...

        var _token;
        var response;
        if (client.scope && (client.scope.indexOf('openid') >= 0)) {
          // An OpenID Connect request
          var _idToken = new IdToken({
            iss: client.redirectUri,
            aud: client.clientId,
            userId: code.userId
          });
          _idToken.save();

          _token = new Token({
            refreshToken: _refreshToken.token,
            idToken: _idToken.sub,
            userId: code.userId
          });
          _token.save();

          // send the token to the consumer
          response = {
            access_token: _token.accessToken,
            refresh_token: _token.refreshToken,
            id_token: _idToken.sub,
            expires_in: _token.expiresIn,
            token_type: _token.tokenType
          };

          res.json(response);
        } else {
          // An OAuth 2 request
          _token = new Token({
            refreshToken: _refreshToken.token,
            userId: code.userId
          });
          _token.save();

          // send the token to the consumer
          response = {
            access_token: _token.accessToken,
            refresh_token: _token.refreshToken,
            expires_in: _token.expiresIn,
            token_type: _token.tokenType
          };

          res.json(response);
        }
      });
    });
  });
});
```

The UserInfo Endpoint

After we have adjusted the Authentication and Token endpoint, the last adjustment we need to handle is adding a new endpoint called UserInfo. The UserInfo endpoint shares the resource owner's profile information with the client. All requests towards this endpoints must be signed using the OAuth 2 Access Token provided as `Authorization` header.

Since we have already written the middleware that handles OAuth 2-based authorization, adding the UserInfo endpoint is as easy as adding this new route:

```
router.get('/userinfo', authorize, function(req, res) {
  // The request got authorized - share profile information
  ...
});
```

This example outlines the power of middleware. Instead of writing duplicate code in order to handle simple tasks like checking the `Authorization` header, we simply mount a middleware that handles this task for all relevant routes.

Session Management with OpenID Connect

OpenID Connect Session Management is a draft[44] that proposes the ability to control End-User sessions to the OpenID Connect stack. This would enable the functionality to not just log the user in (as a lot of OAuth 2 clients) but also handle the termination of sessions, the actual log out process, too.

The specification proposes to pass three parameters to the service provider in order to allow for logging out the user. `id_token_hint` is a required parameter that matches the previously issued ID Token and allows to identify the authenticated End-User plus the user's session. The parameter `post_logout_redirect_uri` will be used for redirects after the logout and is optional. Finally, the parameter `state` can be passed as additional security mechanism - after the logout it will be passed on to the `post_logout_redirect_uri` as query parameter. `state` is an optional parameter, too.

Building an OAuth 2 Client

Client-side OAuth 2.0 varies based on our client's use case and flow. In below sections we will discuss the redirection based Authorization Code flow and how to use the credentials-based flow.

44 *https://openid.net/specs/openid-connect-session-1_0.html*

Using Authorization Codes

Authorization Codes are one of the most common OAuth 2.0 grant types. They find their usage in multiple web- and mobile-applications that leverage redirects in order to exchange the necessary information.

In this example we will implement another Express server that will act as Consumer. The sample will leverage Jade, a Node template engine, and show a simple button that allows to authorize the user.

The first step is enabling session support for Express. In order to do so, execute `npm install express-session --save` to install the required node module and add the following lines to your `app.js` or `index.js` that handles the Express initialization:

```
var session = require('express-session');

app.use(session({
  secret: 'your session secret',
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 60000 }
}));
```

The `resave` option warrants that sessions are saved even if they didn't get modified. By `saveUninitialized` doesn't save new sessions before they are modified - especially when dealing with authentication and authorization purposes it is recommended to disable this option (the default value is `true` according to the documentation[45]).

```
var express = require('express');
var router = express.Router();
var uuid = require('node-uuid');

var SEVER_URL = 'YOUR SERVER URL';
var REDIRECT_SERVER_URL = 'REDIRECT URL';

var CLIENT_ID = 'YOUR_CLIENT_ID';

router.get('/', function(req, res) {
  var state = uuid.v4();
  req.session.state = state;

  var options = {
    url: SERVER_URL + '/authorize',
    client_id: CLIENT_ID,
    redirect_uri: REDIRECT_SERVER_URL + '/callback',
    state: state,
```

45 *https://github.com/expressjs/session#saveuninitialized*

```
    response_type: 'code',
    user_id: 1
  };

  var authorizationURL = options.url +
    '?redirect_uri=' + options.redirect_uri +
    '&user_id=' + options.user_id +
    '&client_id=' + options.client_id +
    '&response_type=' + options.response_type +
    '&state=' + options.state;

  res.render('index', {
    authorizationURL: authorizationURL
  });
});
```

Our Express route will render a template called `index` and will provide the `authorizationURL` to the template in order to avoid hardcoding the Client's details into the template itself.

```
extends layout

block content
  h1 OAuth 2 Client
  a(href="#{authorizationURL}")
    button Authorize
```

After the Service Provider authorized our client, it will redirect to the specified `redirect_uri` and will provide the `state` query parameter (if sent to the server in the previous request) and the Authorization Code itself as `code`.

We leverage `request` - a module that ease making HTTP requests - in order to request the Access Token. Install request by executing `npm install request --save`.

```
var express = require('express');
var router = express.Router();

var request = require('request');

router.get('/callback', function(req, res, next) {
  var state = req.query.state;
  var code = req.query.code;

  // Compare the state with the session's state
  if (state !== req.session.state) {
    next(new Error('State does not match'));
  }

  request.post({
    url: SEVER_URL + '/token',
    form: {
      code: code,
      grant_type: 'authorization_code',
      redirect_uri: REDIRECT_SERVER_URL + '/callback',
      client_id: CLIENT_ID
    }}, function(error, response, body) {
    if (error) {
      // handle the error
      next(error);
    }

    var resp = JSON.parse(body);
    var accessToken = resp.access_token;

    // Use the Access Token for a protected resource request
    ...
  });
});
```

Node is able to perform HTTP requests without leveraging the help of any third-party modules. The reason we chose `request` over implementing requests manually with the `http` module is simplicity and better readability.

Storing tokens on the Client

There is a big discussion about finding the most secure and most-conveniently accessible storage for tokens. Realistically an Access Token expires after a few minutes so we can take that lifetime into consideration. For the Refresh Token on the other hand we deal with a far longer lifecycle and have to handle the token appropriately. From a security standpoint we want to ensure that our tokens are not vulnerable to XSS (Cross-Site Scripting), which means that they shouldn't be accessible from JavaScript (injections) that runs on our client. This reasoning would rule out utilizing the HTML5 Web Storage[46]. The HTML5 Web Storage offers both a local storage that can be accessed via `window.localStorage` and doesn't expire, and `window.sessionStorage`, a session-based storage mechanisms that get's wiped as soon as the user closes the browser's tab.

Classical cookies can be flagged with the `HttpOnly` option, that ensures that the cookie's value can only be accessed from the server-side. This serves as a protection mechanism against XSS-attacks and leaves XSRF/CSRF-attacks for discussion - a big risk that often is not being accounted for. Gladly modern web-frameworks often offer security mechanisms (that might have to be enabled in their configuration) in order to handle the threat of replay attacks and more. We will discuss these concepts in depth in Chapter 6.

46 *http://www.w3schools.com/html/html5_webstorage.asp*

Custom schemes

Especially on mobile platform it has become quite popular to use a custom URL scheme in order to make server to application calls. A URL can be broken down in the following components: `scheme://host/path?query` - this allows apps to define callbacks from the web like `myoauthapp://book/auth?callback`.

Authorization using Resource Owner Credentials or Client Credentials

Next to the Authorization Code Grant Flow another popular OAuth 2.0 Grant is known as the *Resource Owner Password Credentials Grant* as defined in section 4.3 of the OAuth 2.0 specification[47]. This flow represents a simplified way to obtain an Access Token and involves less steps in doing so.

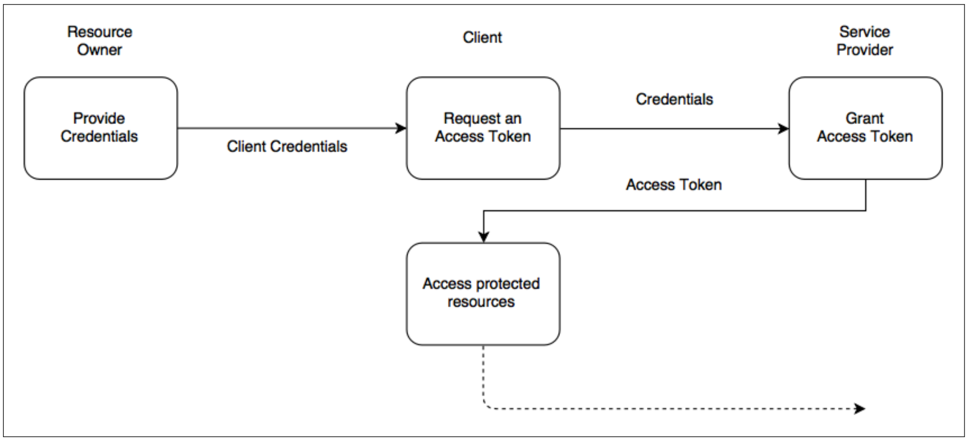


Figure 4-3. OAuth 2.0 Resource Owner Credentials Grant

The specification describes this flow as viable for scenarios, where the user deeply trusts the client - this could be a device's operating system or a highly privileged application.

The Resource Owner Credentials Grant Type is very similar to the Client Credentials Grant Type, in which the Client provides its credentials to the Service Provider. The difference is that in the first case the user trusts the client and therefore submits his details, whereas in the latter example the client owns the resources himself and therefore another step in the authentication scenario can be removed:

47 *https://tools.ietf.org/html/rfc6749#section-4.3*

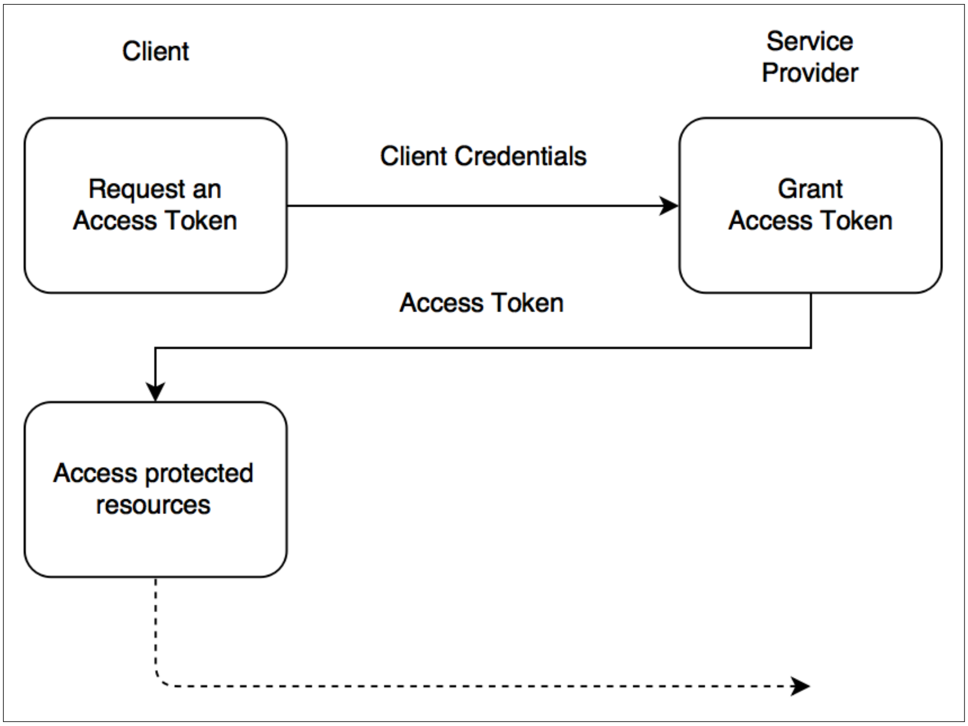


Figure 4-4. OAuth 2.0 Client Credentials Grant

As you might have guessed, in this flow the client simply asks for the user's credentials. A popular example for a Client Credentials driven API is PayPal's REST API which uses client credentials in order to authorize a merchant's application to accept payments[48].

Adding OpenID Connect functionality to the client

Since OpenID Connect heavily relies on OAuth 2.0 in terms of communication and security mechanisms only few changes or additions to our client-side OAuth 2.0 example are needed in order to add support for OpenID Connect.

OpenID Connect provides two ways in order to handle the consumer and service provider communication. The first flow is known as *Basic Client* and is being detailed in the Basic Client Implementer's Guide[49], the second flow, the *Implicit Client* is

48 *https://developer.paypal.com/docs/api/#authentication—headers*

49 *http://openid.net/specs/openid-connect-basic-1_0.html*

detailed in the Implicit Client Implementer's Guide[50]. The implicit flow is known as client-side flow or implicit grant flow and removes the need for an authorization token - the access token is simply sent back to the server and no refresh token is being issued. This flow is generally seen as less secure than the basic flow and will therefore not be outlined in detail. A great post[51] covering the OAuth 2.0 implicit flow has been written by Sven Haiges.

The OpenID Connect Basic flow

When dealing with OpenID Connect's Basic flow we'll need to adjust the client-side OAuth 2.0 requests for authentication and accessing resources. For OAuth 2.0 the `scope` parameter is optional and needs to be used when added to the initial authentication request. When adding OpenID Connect functionality to the client, `scope` becomes a required parameter that needs to be set to cover at least `openid` as scope value[52].

Table 4-4. OpenID Connect request scope values

`scope value`	necessity	description
`openid`	required	Specifies that the Client is making an OpenID Connect request
`profile`	optional	Access to the User's profile Claims such as: `name`, `family_name` and `given_name`
`email`	optional	Access to the `email` and `email_verified` Claims
`address`	optional	Access to the `address` Claim
`phone`	optional	Access to the `phone_number` and `phone_number_verified` Claims
`offline_access`	optional	Request that an OAuth 2.0 Refresh Token is issued in order to allow for obtaining a new Access Token and therefore allowing to access the UserInfo endpoint even when the user is not present.

The request in order to handle the client's authentication will be altered to this form:

```
var options = {
  url: SERVER_URL + '/authorize',
  client_id: CLIENT_ID,
  redirect_uri: REDIRECT_SERVER_URL + '/callback',
  state: state,
  scope: 'openid',
  response_type: 'code',
  user_id: 1
};
```

50 *http://openid.net/specs/openid-connect-implicit-1_0.html*

51 *https://labs.hybris.com/2012/06/05/oauth2-the-implicit-flow-aka-as-the-client-side-flow/*

52 *http://openid.net/specs/openid-connect-basic-1_0.html#RequestParameters*

```
var authorizationURL = options.url +
  '?redirect_uri=' + options.redirect_uri +
  '&user_id=' + options.user_id +
  '&client_id=' + options.client_id +
  '&scope=' + options.scope +
  '&response_type=' + options.response_type +
  '&state=' + options.state;

res.render('index', {
  authorizationURL: authorizationURL
});
```

The difference compared to the OAuth 2 client is that this time we are required to pass the `scope` parameter (set to at least `openid`) in order to match the specifcation's minimum requirements.

After obtaining the user's authorization, the authorization code is issued and exchanged for the actual token. The token endpoint's response will differ slightly and will also contain an `id_token` attribute:

```
{
  "access_token": "71518132-d27b-4828-9317-5571a46c89fb",
  "refresh_token": "3ae3e757-7c32-492d-8af5-8dba943d2ec3",
  "id_token": "ee0b16a5-5be7-4629-8d1b-bf3fd7ea64a9",
  "expires_in": "10800",
  "token_type": "bearer"
}
```

The ID Token can be used as an additional security mechanism. It contains claims about the authentication of an end-user as defined in the OpenID Connect specification[53]. An example value that can be easily validated is `azp` (authorized party) that must match the Consumer's `client_id`.

The OpenID Connect specification provides a list of all 21 standard Claims[54] that can be returned by the server's UserInfo endpoint. Claims must be returned in JSON format unless the format was defined differently during the client's registration[55].

Beyond OAuth 2.0 and OpenID Connect

In this chapter we have discussed the ins and outs of both OAuth 2.0 and OpenID Connect. Both protocols find strong adoption within the industry and empower millions of users and a multitude of applications. Still, with the rise of mobile apps and especially mobile authentication and authorization, the need to provide better User

53 *http://openid.net/specs/openid-connect-core-1_0.html#IDTokenValidation*

54 *http://openid.net/specs/openid-connect-basic-1_0.html#StandardClaims*

55 *http://openid.net/specs/openid-connect-registration-1_0.html*

Experience is even more prominent than on the desktop. In the following chapter we will discuss current multi-factor authentication systems and viable alternatives, such as biometrics, to identify users and grant authorization towards certain actions or information.

CHAPTER 5

Alternate methods of identification

Tim Messerschmidt and Jonathan LeBlanc

Due to the heavy intersection between mobile devices, desktop clients and a new breed of connected hardware out of the Internet of Things section, the demand for a new class of authentication and authorization technology is on the rise. In this chapter we will discuss upcoming standards like FIDO[1] that enable covering multiple form factors and are able to scale beyond software-based authentication technology.

Device and browser fingerprinting

Next to regular authentication and authorization scenarios, device and browser fingerprinting allows for a more passive way to identify users across a big target group. Applications like Am I Unique?[2] are broadly available and can leverage many different factors in order to decide if a user is unique or not.

When performing device and browser fingerprinting the user is usually tested against some very general and broad factors - such as the device's platform, the current browser or if cookie's are enabled on the device - and then against more granular and subtle determinants, like the device's resolution, timezone, the browser's enabled plu-

1 *http://fidoalliance.org*

2 *http://amiunique.org*

gins or user agent. When having Flash enabled, services like Am I Unique? or Panopticlick[3] are even able to obtain a list of currently installed fonts.

In total there are 8 different factors that can be concatenated and lead to a browser's fingerprint:

Table 5-1. Browser measurements to determine uniqueness

variable	obtained through
`User Agent`	HTTP
`HTTP ACCEPT headers`	HTTP
`Cookies enabled`	HTTP
`Screen resolution`	AJAX
`Timezone`	AJAX
`Browser plugins`	AJAX
`System fonts`	Flash or Java applets, collected through AJAX
`Supercookie test`	AJAX

Additional factors, such as the user's geolocation, can be obtained through HTML5 if the user agrees to share them or by analyzing the user's IP address (which does not involve the need for the user's consent).

Panopticlick released a paper on browser uniqueness that is a very worthwhile read and a great source for further information on this subject[4].

2-factor authentication and n-factor authentication

Due to the known weaknesses and issues that come along with basic authentication through passwords, the demand for more secure login methods is high. 2-factor authentication (2FA) relies on the addition of another token, such as a One-Time Password, which is consumed after usage and therefore prevents common security exploits, such as replay attacks. In this section we aim at explaining the basic concepts of both 2-factor authentication and the upcoming n-factor authentication technologies.

3 *http://panopticlick.eff.org*

4 *http://panopticlick.eff.org/browser-uniqueness.pdf*

n-factor authentication

n-factor authentication , also known as multi-factor authentication (MFA), revolves around the concept of assuming three basic components that every individual should have:

1. Something you know
2. Something you have
3. Something you are

When examining these three requirements you will quickly realize that they match concepts, that we have discussed before. *Something you know* is the most basic component and can be assumed as granted: it can be as simple as a password or passphrase.

The second item on our list, *Something you have*, is an additional component, that aims at securing our password or passphrase by adding another layer of protection. A popular example are smart cards or RSA Tokens, that are being used for RSA's SecurID[5] network authentication technology. As of 2014 about 1.75 billion people worldwide will have access to mobile phones[6] - a small and affordable piece of technology that can act very easily as additional physical layer in authentication and authorization technology. By being able to receive text messages and/or emails and allowing for the installation of authentication applications that generate one-time passwords , such as Google Authenticator[7] and Authy[8], people are able to secure existing logins.

Lastly, *Something you are*, focusses on the individual's identity and adds a handful of new challenges, that we will discuss in the following section. The basic assumption here is that the usage of something intrinsic, such as the individual's fingerprint, allows to uniquely identify the user amongst all users and therefore adds a third layer of security.

One-Time Passwords

One-Time Passwords, known as OTP, have been positioned in the industry as a means to fight traditional password weaknesses and exploits. By being ever-changing and only usable once, they reduce an application's attack surface drastically.

5 *http://www.webopedia.com/TERM/R/rsa_secure_id.html*

6 *http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536*

7 *http://github.com/google/google-authenticator*

8 *http://www.authy.com*

Currently, there are three different ways to generate One-Time Passwords: The first implementation, time-synchronization, generates short-lived tokens. Popular 2-factor authentication applications, such as Authy or Google Authenticator, use this method to generate OTPs.

Both the second and third implementation are based on mathematical algorithms and generate long-lived tokens: One way to handle these OTPs is generating them based on the previous password and therefore require to be used in a predefined order. The other way to handle mathematically generated OTPs is generating them based on a random challenge.

When not being generated by a client-side application, OTPs can be delivered by either text messages or emails. The industry tends to favor text over email at the moment since it's broadly available, a phone number is rated to be unique across all users, and can be made accessible through text-to-speech and therefore also cover landline phones. A reason to use emails instead is the cost of sending a text message and the inability to check if the text message arrived at it's destination. Another issue of text messages is the weak (A5/x[9]) or non-existant encryption standards that allow for man-in-the-middle attacks.

On mobile devices the usage of emails in order to transport One-Time Password has one big advantage for User Experience: applications can automatically open and import the OTP - this heavily reduces friction and is being used by companies like Slack. The key to automating this process is registering a custom url-handler (via Android's application manifest[10] and utilizing URL Schemes on iOS[11]) that detects when URLs of a certain markup are handled.

9 *http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2006/CS/CS-2006-07.pdf*

10 *http://stackoverflow.com/a/2448531/636579*

11 *http://code.tutsplus.com/tutorials/ios-sdk-working-with-url-schemes—mobile-6629*

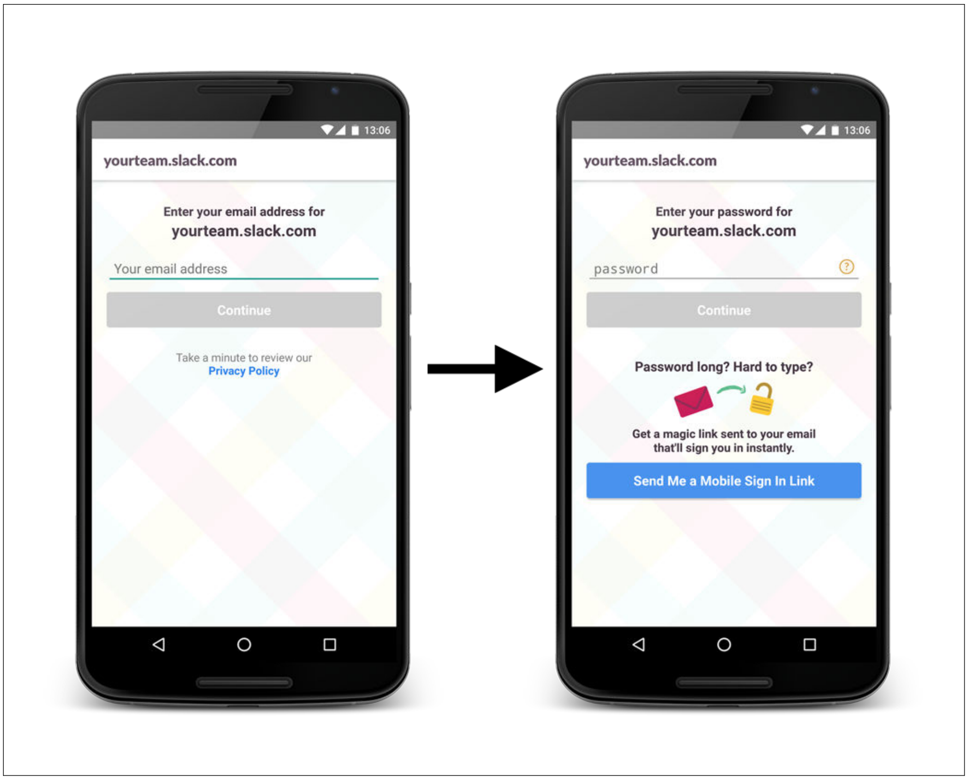
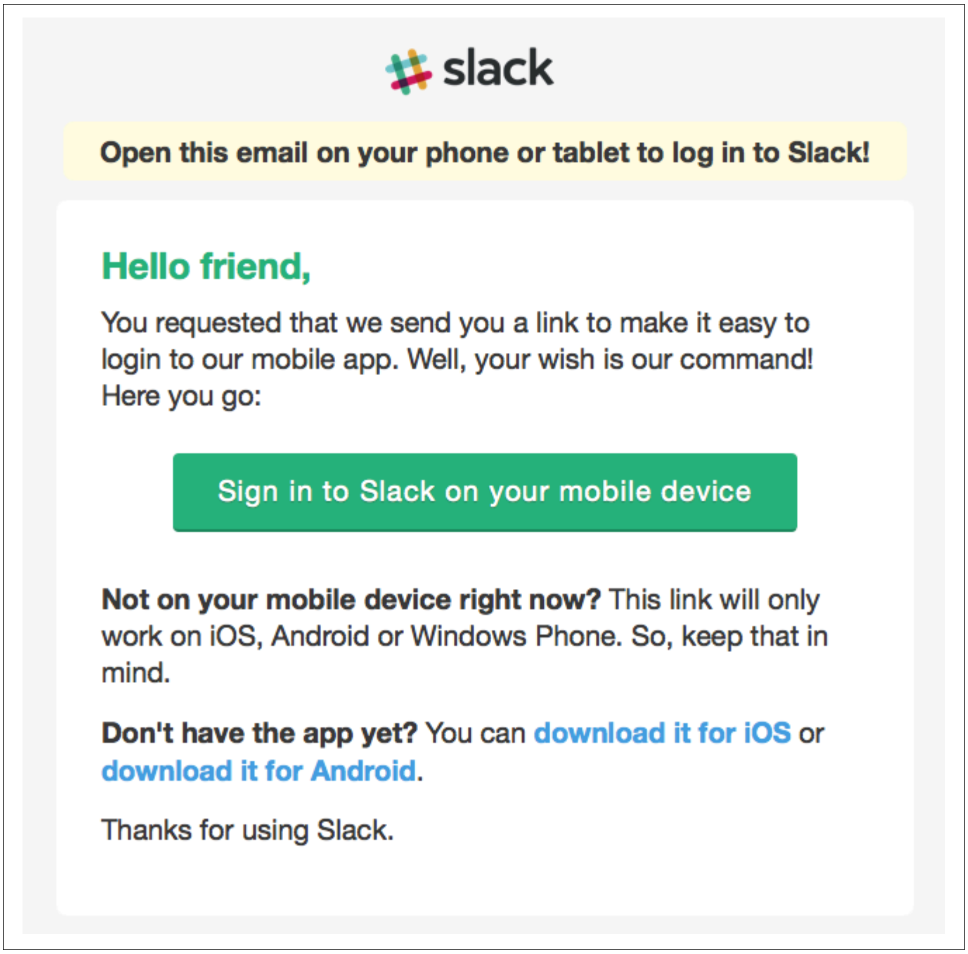

13:06
yourteam.slack.com
Enter your email address for
yourteam.slack.com
Your email address
Continue
Take a minute to review our
Privacy Policy
13:06
yourteam.slack.com
Enter your password for
yourteam.slack.com
password
Continue
Password long? Hard to type?
Get a magic link sent to your email
that'll sign you in instantly.
Send Me a Mobile Sign In Link

Figure 5-1. Slack's mobile sign-in flow

This sign-in flow results in an email like the following:



Open this email on your phone or tablet to log in to Slack!

Hello friend,

You requested that we send you a link to make it easy to login to our mobile app. Well, your wish is our command! Here you go:

Not on your mobile device right now? This link will only work on iOS, Android or Windows Phone. So, keep that in mind.

Don't have the app yet? You can download it for iOS or download it for Android.

Thanks for using Slack.

Figure 5-2. Slack's sign-in email for mobile devices

When analyzing the email's source code you'll notice that a URL like the following is used: `https://slack.com/z-app-211= 9624547-19991285158-cJ1DJfifFa?s=3Dslack`. Since Slack clearly owns the authority over slack.com (and no other applications *should* claim any URI's containing this domain), no custom scheme workaround like `myapp://auth.com/` is needed. By clicking on the "Sign in to Slack on your mobile device"-button from within your mobile email client, the Slack application will open up and the user will be signed in.

Since version 6.0 (Marshmellow), Android allows to declare website associations[12]. This mechanism helps to protect your native applications by preventing third-party applications from accessing information that are meant for internal consumption only.

Implementing two-factor authentication with Authy

Now that we understand how OTPs work, let's see how we can implement these within our own apps and websites. For this example, we're going to use a Twilio authentication service called Authy[13]. Authy will allow us to do a number of things that we'll need for a 2FA system, such as:

- Register / delete 2FA user accounts on our service.
- Send SMS verification codes to those users.
- Verify the verification codes once the user enters them on the website to verify themselves.

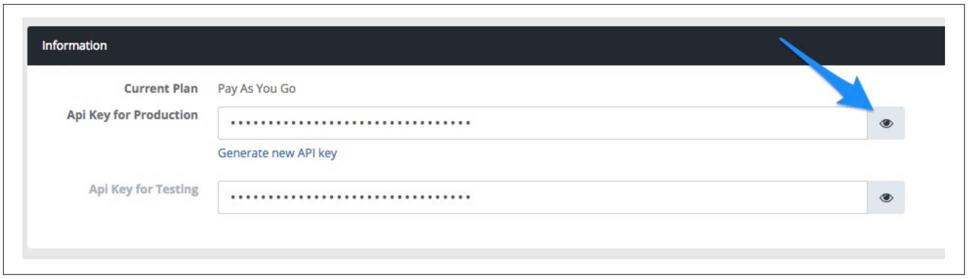
With that said, our first task will be to set up an application with Authy and get a key that we will use to verify our application against the service. We can do that be following the steps below:

- Go to the Authy dashboard at *https://dashboard.authy.com/signin*.
- Sign in or register a new account.
- Click to "Access Authy Dashboard" (direct link).
- Click on "Enable Two-Factor Authentication", and select your preferred verification method, which is required to create a new application.
- Click on "Create your first app".
- Enter an application name on the form that pops up, then hit "Create". Follow the rest of the instructions to create the application.

Once the application dashboard comes up, at the top of the page you will see a section for information, which will include your hidden product and sandbox keys. We're going to be using our production key, so click the eye beside the hidden key to reveal it.

12 *http://developer.android.com/training/app-links/index.html#web-assoc*

13 *https://www.authy.com/*


Information

Current Plan: Pay As You Go

Api Key for Production: ••••••••••••••••••••••••••••••••

Generate new API key

Api Key for Testing: ••••••••••••••••••••••••••••••••

Figure 5-3. Authy key details

Take note of that key, because we'll be using it in our Authy 2FA example.

The complete sample code for the below Authy example is available at *https://github.com/iddatasecuritybook/chapter5/tree/master/authy-2fa*.

With our key in hand, let's dive into a practical walkthrough of how to implement 2FA using the service. First we're going to need to install a few npm modules, specifically:

- **body-parser**: For dealing with JSON / URL -encoded payloads post Express 4.0.
- **authy**: A helpful module for working with authy functionality for users and tokens.

We can pull down these packages via npm with the following terminal commands:

```
npm install body-parser --save
npm install authy --save
```

We can now create our .js file, and instantiate our packages and the body-parser functionality.

```
var express = require('express'),
    app = express(),
    bodyParser = require('body-parser'),
    authy = require('authy')('YOUR AUTHY PRODUCTION API KEY');

app.use(bodyParser.json());                         //to support JSON-encoded bodies
app.use(bodyParser.urlencoded({ extended: true }));     //to support URL-encoded bodies
```

In the above, we set up our express, body-parser, and authy variables. With the Authy instantiation, we pass in that Authy production key that we obtained when we create our application on the site. In the last two lines we then set up body-parser to be able to handle JSON and URL encoded objects that we will need to parse from our requests later.

With Express, we can now set up a few routes in our application to handle POST requests to different endpoints for working with user setup and token verification. Let's start with defining a route to handle user registration.

When should you register a new user with Authy?

The Authy registration for new users should be done when you have a new user creating an account with your site or service, during your regular registration process. As you are storing user information for your site, you will also store the user id that Authy provides during registration.

```
//route: register a new user via provided email and phone number
app.post('/register', function(req, res){
    var email = req.body.email;
    var number = req.body.number;

    authy.register_user(email, number, function (err, response){
        //expected response:
        //{ message: 'User created successfully.',
        //  user: { id: 16782433 },
        //  success: true }
        res.send(response);
    });
});
```

The above route will allow any POST request sent to the /register endpoint of the application to be handled. The endpoint is expecting two pieces of data to be sent to it:

- **email**: The email of the user to be registered. This should match the email that was stored in your user records when you registered the individual.
- **number**: The phone number of the user to be used for the SMS 2FA process.

With that information obtained, we then make a request to *authy.register_user(…)*, passing in the email and number that we just pulled from the POST body. If all was successful, the return value (stored in response), should contain three pieces of data:

- **message**: The human readable success message.
- **user**: The user ID of the newly registered user. This should be stored in your user database for sending the 2FA requests.
- **success**: A boolean true / false indicating the success state.

Next on our list is to set up the ability to send SMS 2FA messages to a given user ID.

When should you send the SMS verification code?

SMS verification should be conducted during login. When a user enters their first set of credentials (typically username / password), you can then send the SMS message from Authy for a second level of authentication.

```
//route: send authy SMS message with verification code
app.post('/sms', function(req, res){
    var uid = req.body.uid;

    authy.request_sms(uid, function (err, response){
        //expected response:
        //{ success: true,
        //  message: 'SMS token was sent',
        //  cellphone: '+1-XX12362760' }
        res.send(response);
    });
});
```

The route above will accept any POST request to the */sms* endpoint, and will expect one piece of data to be POSTed:

- **uid**: The user ID that was obtained from registering the user with Authy, during the last step.

Once we pull out that value, we can then make a request to *authy.request_sms(…)*, passing along that uid and a callback. This will attempt to send an SMS verification code to the phone number that is registered for that given user during the registration step. In the response object (on success), we are expecting a few parameters:

- **success**: A boolean true / false indicating the success state.
- **message**: The human readable success message.
- **cellphone**: The cell phone number that the SMS was transmitted to.

At this point the user has obtained a verification code, they enter the code on your site, and you will need to verify that it is correct.

How and when should you validate a verification code?

When the user is sent the SMS verification code during the login step (for second factor verification), you should supply a method for them to enter in the code that they see on their mobile device on your site.

```
//route: verify a provided verification token against a given user
app.post('/verify', function(req, res){
    var uid = req.body.uid;
    var token = req.body.token;

    authy.verify(uid, token, function (err, response){
        //expected response:
        //{ message: 'Token is valid.',
        //  token: 'is valid',
        //  success: 'true' }
        res.send(response);
    });
});
```

The above route will handle the verification step. It will accept a POST request to the */verify* endpoint, and will expect two pieces of data in the POST body:

- **uid**: The user ID that Authy provided during the registration step.
- **token**: The verification token that the user was sent via SMS during the last step.

Once we obtain that information, we can then call *authy.verify(…)*, passing in the uid, token, and a callback function. If the verification step completes successfully, we will be expecting three pieces of data coming back from the response:

- **message**: The human readable success message.
- **token**: Verification whether the token is valid or not.
- **success**: A boolean true / false indicating the success state.

Once we verify the token is legitimate, we can then allow the user to enter the site, and the 2FA process is now complete.

The last step we may need is when a user deletes their account from our site. We want to ensure that we cleanup all residual user information, including their Authy user registration data.

When should you delete users from Authy?

When a user deletes their account with your site or service, you should also clean up their information in Authy by deleting the registered account. The registration / deletion steps should be synced with your site / service registration and deletion steps.

```
//route: delete an existing user
app.post('/delete', function(req, res){
    var uid = req.body.uid;
    authy.delete_user(uid, function (err, response){
        //expected response:
```

```
        //{ message: 'User was added to remove.', success: true }
        res.send(response);
    });
});
```

The route above will accept a POST request to the */delete* endpoint, and expect one item in the POSTed data:

- **uid**: The user ID that Authy provided during the registration step.

When obtained, we then make a call to *authy.delete_user(…)*, passing along the user ID, and a callback. If the deletion was successful, we should see the following parameters come back in the response:

- **message**: The human readable success message.
- **success**: A boolean true / false indicating the success state.

Once done, the user has been removed from the Authy registration system. In our app sample, the last thing we need to do is start the server.

```
app.listen(process.env.PORT || 3000);
```

This will listen on the expected port (such as if running via Heroku), or on port 3000 otherwise. Once the server is up and running (assuming on localhost port 3000 in this case), we can then run some tests by sending POST requests from the terminal to each of the endpoints we set up.

First we issue a request to register a new user.

```
curl -H "Content-Type: application/json" -X POST -d
'{"email":"jenny@email.com, "number":"18675309"}' http://localhost:3000/register
```

We send an HTTP POST request to the register endpoint, passing along an email and phone number in the POST body. The JSON response from that will give us the user ID for the newly registered person, which we will use for the next step.

The second step is to trigger the send of an SMS to the phone of that registered user.

```
curl -H "Content-Type: application/json" -X POST -d
'{"uid":"16572253"}' http://localhost:3000/sms
```

From the registration request we obtained a user id from Authy. We send that UID through to the SMS endpoint. The response from that should be a text message showing up on the phone number registered. The response from that will provide us with a verification code via SMS.

Next, we send the token we have in our SMS through for verification.

```
curl -H "Content-Type: application/json" -X POST -d
'{"uid":"16572253", "token":"0512278"}' http://localhost:3000/verify
```

We send the user ID and token via a POST body to the verify endpoint, which should provide us with a message stating that the token is valid, if the request was successful.

The last step is to clean up the user records by deleting the user we just created

```
curl -H "Content-Type: application/json" -X POST -d
'{"uid":"16572253"}' http://localhost:3000/delete
```

We send the user ID to the delete endpoint which, on success, will provide us with a success message response.

With all of that in place, we now have the structure in place to provide 2FA SMS token verification for our users.

Biometrics as username instead of password

With the growing availability of fingerprint scanners on mobile devices, such as the iPhone device family and newer Android devices, more and more applications try to identify use-cases that enhance the overall user experience. This surge in new technology led to people starting to ask for using their fingerprint in order to replace password prompts on their phones.

From a logical standpoint it might occur as an easy choice to leverage biometrics in order to authorize access to applications, unlock a device's screen, and much more, but effectively new issues are being provoked. Passwords are traditionally vulnerable, as we have discussed in Chapters pass[<a data-type="xref" data-xrefstyle="select:labelnumber" href="#ch1">#ch1</a> and <a data-type="xref" data-xrefstyle="select:labelnumber" href="#ch2">#ch2</a>], and can be leaked or exposed to third parties. When using simple passwords we can simply alter that password and exchange it for a new, more secure password. When using fingerprints we run into a whole new dimension of issues: human beings have a maximum of ten fingers and it is highly desirable that those fingerprints are not invalidated by being exposed towards the public.

Using fingerprints as a security mechanism

The German Chaos Computer Club managed to bypass the security mechanisms of Apple's TouchID in 2013. By replicating a fingerprint using a high resolution photograph, the CCC managed to trick an iPhone 5s's TouchID sensor into unlocking the phone.

In the exploit's summary[14], the CCC highly recommends not to use fingerprints in order to secure anything.

14 *http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid*

Tim Bray, co-inventor of XML and contributor to the IETF's OAuth work, expressed his opinion about using fingerprint scanners and other biometric factors in a blog post[15] that led to an interesting discussion[16] with John Gruber, inventor of the Markdown standard. Gruber states that using a fingerprint is still better than using no security (like not locking your phone with a 4-digit pin or a passphrase) or weak security.

Considering the discussion between Bray and Gruber and the fact, that the CCC managed to exploit fingerprint scanners into unlocking, it might be wise to consider biometric factors less as a security mechanism and more as a mechanism in order to prove identity.

How to rate biometric effectiveness

When handling biometric factors for authentication scenarios the false-positive rate, also known as false-acceptance rate, of the used mechanism is critical. Google requires third-party manufacturers that want to implement fingerprint scanners for Android phones to use an implementation that leads to a false-positive rate of not higher than 0.002%[17]. False-rejection, another confounding factor, leads to user frustration and is to be avoided - Google's guidelines defines a maximum rate of 10% and a maximum latency between scan and action of 1 second. A third important criteria to secure fingerprint scanning is limiting the number of false attempts before disabling fingerprint scanning; Apple allows for 3 false attempts on iOS devices before asking the user to unlock the phone differently, while Google defines a maximum of 5 tries before disabling fingerprint scanning for at least 30 seconds (per the manufacturer guideline).

Face recognition

Facial recognition aims at using either digital images or videos in order to identify people. In order to do so a wide array of so-called landmarks and features are being extracted and processed in order to match profiles. Factors such as the relative position and size of those landmarks are being normalized and compared using either geometric (compare distinguishing features) or photometric (generating statistical values) approaches. Emerging three-dimensional recognition systems have proven to be less sensitive towards changes in lighting and can help with improving recognition by allowing to scan different angles (often at the same time by stacking multiple sensors on the same chip).

15 *https://www.tbray.org/ongoing/When/201x/2013/09/22/The-Fingerprint-Hack*

16 *https://twitter.com/gruber/status/381857717406560257*

17 *https://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cdd.pdf*

Various banks such as the National Bank of Costa Rica and Ecuador have announced to use facial recognition technology on mobile devices to secure access to banking accounts[18]. Alipay, the Alibaba Group's online payment platform, announced in November 2015 to roll out facial recognition to both iOS and Android devices[19]. Both examples demonstrate that the finance industry does not seem to be completely behind fingerprint technology and tries to evaluate other biometric factors on a broader scale.

Retina and Iris scanning

In a similar fashion to face recognition, retina scans rely on identifying unique patterns. When observing a person's eye, blood vessels can be analyzed and used in order to identify users. Even identical twins do not share the same blood vessel network and therefore cannot circumvent this security concept[20].

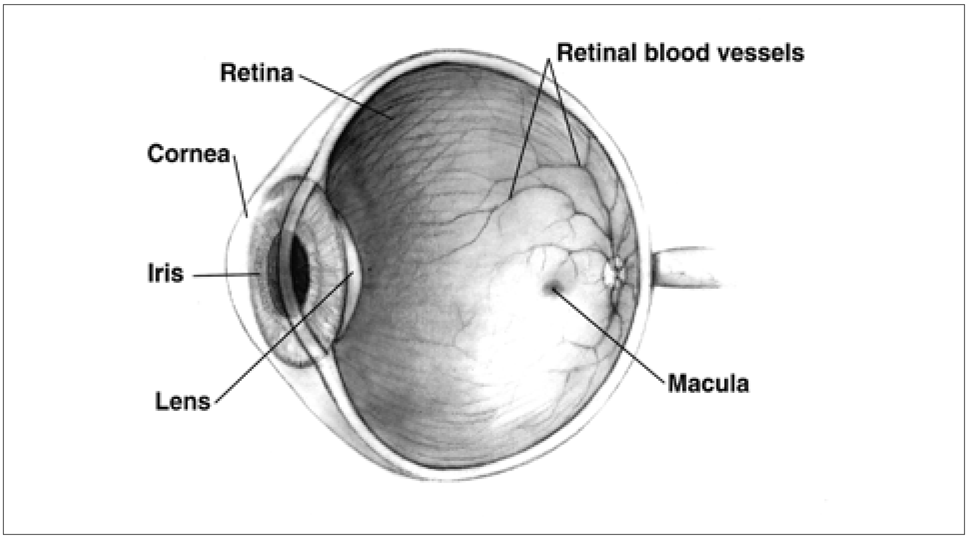


Figure 5-4. Anatomy of an Eye - illustration taken from the National Eye Institute[21]

While both retina and iris scanning use cameras to identify people, the key difference lies in the identification process itself. While retina scans rely on light being absorbed by blood vessels in order to analyze a person's retina, iris scanning is based on taking

18 *http://www.biometricupdate.com/201601/facephi-facial-recognition-solution-to-authenticate-banco-nacional-of-costa-rica-clients*

19 *http://findbiometrics.com/alipay-facial-recognition-comes-to-ios-android-212227/*

20 *http://blog.m2sys.com/biometric-hardware/iris-recognition-vs-retina-scanning-what-are-the-differences/*

21 *https://nei.nih.gov/health/macularhole/macularhole*

an image of an eye that will be analyzed in order to identify structure. These images can be captured from a distance of 3 to 10 inches and therefore are considered less intrusive than retina scans that require the the user's eye to be much closer to the scanning device. An iris is supposed to have 266 unique spots that can be leveraged to determine uniqueness[22].

While a person's retina might change due to temporal or permanent effects (like diabetes or high blood pressure), the iris supposedly stays the same between birth and death of a human being.

Vein recognition

While fingerprints remain usable as long as they can be duplicated or obtained in any other way, a persons veins are only viable for authentication mechanisms as long as blood flows through the body. Fujitsu has deployed palm vein recognition as a solution across ATMs in Japan that leverages biometric details to encrypt the dataset itself and therefore removes the need for encryption keys[23].

Upcoming standards

When analyzing the current authentication and authorization standards, it becomes quite apparent that the industry has not decided on a common standard. In this section we'd like to present three currently viable contenders with very different focus and industry backing: The FIDO Alliance, Oz, and the Blockchain.

FIDO Alliance

The FIDO Alliance - which stands for Fast Identity Online - is a new industry alliance between major contributors such as Google, Blackberry, Microsoft, PayPal and Lenovo. FIDO aims at providing a scalable identity solution that covers multiple platforms and covers the three basic principles of authentication - *something you have*, *something you know*, *something you are* - by providing two different scenarios: Universal Authentication Framework (UAF) and Universal 2nd Factor (U2F).

Both U2F and UAF are compatible with current federated identity services - such as OpenID and SAML - or authorization protocols like OAuth.

22 *http://www.globalsecurity.org/security/systems/biometrics-eye_scan.htm*

23 *http://www.biometricupdate.com/201510/fujitsu-laboratories-develops-method-to-convert-biometric-data-into-cryptographic-key*

UAF

UAF got designed with passwordless and mutli-factor authentication flows in mind. By leveraging local mechanisms - such as using microphone input, entering a PIN or fingerprint-scanning - a trust relation is established. The beauty of the protocol is, that various factors can be combined; this kind of *security-layering* is a concept that will be outlined in the sixth chapter of this book.

From a privacy perspective, the FIDO alliance dictates that as minimal data as needed is collected and only used for FIDO purposes. User verification is handled locally on the device and does not convey any biometric details to third parties.

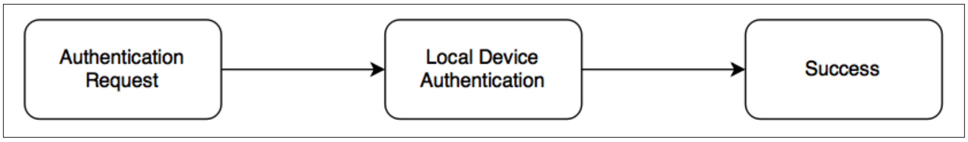


Figure 5-5. FIDO UAF Authentication

U2F

While UAF aims at combining various factors towards a secure and passwordless solution, U2F aims at augmenting an existing authentication implementation by adding a second factor. The second factor allows to simplify passwords requirements to 4-digit PIN codes and manifests in a device that presents the second factor via USB or NFC. This piece of hardware is usable across all implementing online services as long as the web-browser supports the U2F protocol.

The devices are supposed to be designed with mainstream adoption in mind. This is why the design principles are very minimal and allow for affordable hardware that can be distributed widely. From a security perspective a secure key will be provided that will be provided to manufacturers of secure elements and will change with every chipset batch.

U2F got designed with flexibility in mind: Multiple persons can share one device and each person can use multiple devices to secure accounts across implementing sites.

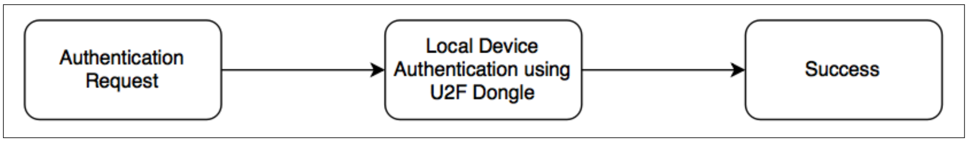


Figure 5-6. FIDO U2F Authentication

U2F utilizes a special registration and authentication message format in order to communicate with all supporting devices and browsers. The authentication message is defined as the following:

Table 5-2. Authentication message format

parameter	description
Control byte	`0x07` to check if the key handle was created for the provided application parameter, `0x03` if a real signature and user presence is required
Challenge	SHA-256 hash of client data (stringified JSON)
Application	SHA-256 hash of the application identifier
Key handle length byte	Defines the length of the following key handle
Key handle	Provided by the relying party & obtained during registration

In case of a successful authentication, the response contains a parameter that provides information about the user presence, a counter that increments whenever a successful authentication operation was performed and a signature consisting of:

- application parameter (32 byte)
- user presence byte (1 byte)
- counter (4 byte)
- challenge parameter (32 byte)

Oz

Eran Hammer - known for his contributions to both OAuth 1.0 and 2.0 - recently (September 2015) published a web authorization framework called Oz[24], that aims at compiling industry best practices in order to provide not just a protocol but rather a concrete implementation that is very opinionated about details such as client-side cryptography using HMAC.

This framework does not try to be a solution that covers all platforms and form factors but rather a viable tool for JavaScript-based applications that aim at implementing a secure solution for authorization.

Currently Oz provides a OAuth 1.0-esque authorization flow and is based on two current solutions: Hawk, a client-server authorization protocol[25], and Iron - a tool that allows encoding and verifying of JavaScript objects [26]. Opposed to OAuth, Oz tries not to to handle user authentication - it's sole purpose is handling application-to-server-authorization scenarios. From an architecture standpoint Oz is very similar to a slimmed-down implementation of OAuth 2 enriched with security best practices.

24 *http://github.com/hueniverse/oz*

25 *https://github.com/hueniverse/hawk*

26 *https://github.com/hueniverse/iron*

The Blockchain

Developed to verify Bitcoin transactions, the blockchain is slowly becoming a powerful tool beyond the scope of cryptocurrency and the payments landscape.

The idea behind using the blockchain for identity scenarios is simple: a user can store proof of certain attributes - such as first and family name, address or date of birth - and make the cryptographic hash of these attributes publicly available to anyone that is able to provide the user's public key. This allows individuals to verify information, while authenticity of these details can be ensured. The interesting twist in this concept is the ability to decide which information are supposed to share.

Let's use the example of a car accident - somebody scratched your car and wants to provide you important information such as insurance details, contact name and phone number. Currently we'd hope that the person gave us the correct details and cannot verify anything until it is probably too late. Utilizing the blockchain, we'd rely on exchanging cryptographic hashes and could verify all information provided on the spot.

A company called ShoCard[27] tries to build upon this concept by providing a consumer-friendly mobile application. All information are being stored in the public blockchain data layer[28] and made accessible on demand.

27 *http://www.shocard.com*

28 *http://www.shocard.com/what-is-a-blockchain*

CHAPTER 6

Hardening Web Applications

Tim Messerschmidt

After we have spent some time in the fourth chapter in order to discuss the relevance of OAuth 2.0 and OpenID Connect and analyzed the relevance of identity and biometry plus the impact of multi-factor authentication, this chapter will cover security considerations for Node applications - especially focussing on Express.

In security it is fantastic to provide a secure solution to identify users and authorize access towards certain resources, but due to a multitude of different attack vectors it is simply not enough to rely on a simple protocol in order to secure our application. This is why we tend to go for layered security. Let's use a simple analogy: In medieval times a castle was secured by a gate - which was a good way to keep people out and defend property. When being combined with stone walls instead of simple wood the gate got even better, since even heavy machinery needed more time in order to breach the walls. If combined with a moat, the wall and gate became even more useful. If we take this analogy and apply it to today's standards, we want to make sure that our application is not just capable of identifying users, but also ensures that we withstand a DDoS attack, secure our sessions, and prepare for potential XSS and CSRF attacks.

Securing sessions

Sessions have a very simple reason to exist: being able to persist user logins across various routes within our application without the need to re-authenticate the user by prompting for usernames and passwords over and over again. Just like passwords, sessions have to meet certain requirements in order to meet security expectations: session ids should be unique, non-guessable and non-sequential. Just like passwords, sessions benefit from long session ids that decrease attack vectors.

Different types of sessions

Before we dive into the details of sessions and securing sessions on top of the mechanisms that Express provides by default, it is sensible to explore the difference between cookies and sessions and what makes them interesting first.

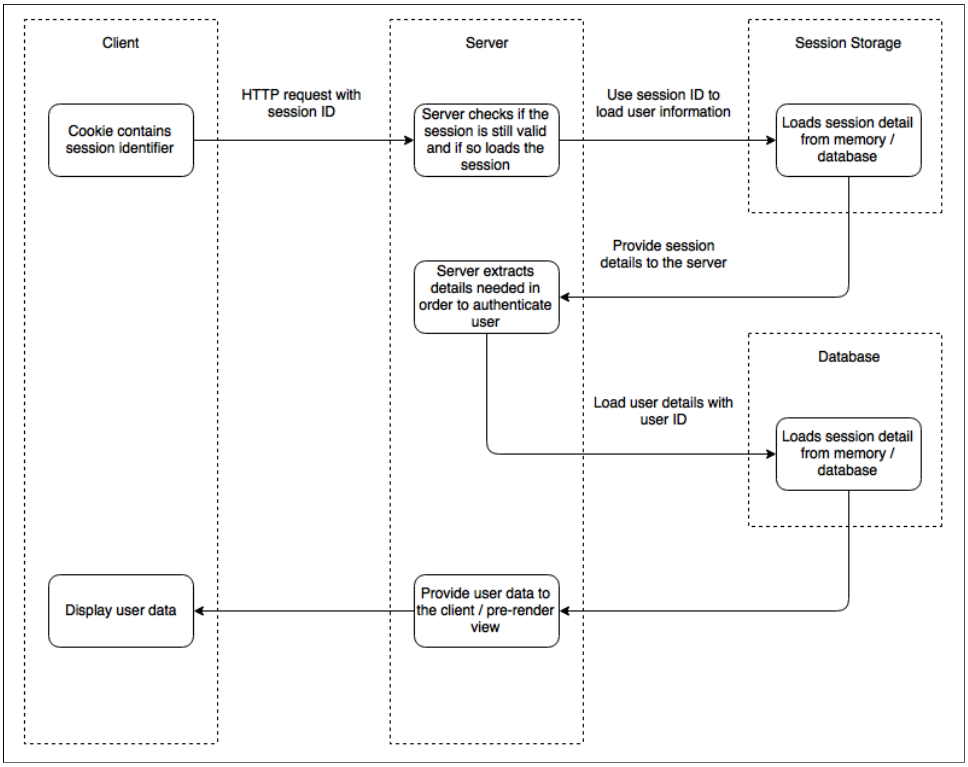

Client
Server
Session Storage
HTTP request with session ID
Cookie contains session identifier
Server checks if the session is still valid and if so loads the session
Use session ID to load user information
Loads session detail from memory / database
Provide session details to the server
Server extracts details needed in order to authenticate user
Database
Load user details with user ID
Loads session detail from memory / database
Display user data
Provide user data to the client / pre-render view

Figure 6-1. Session & Cookie Relationship

What's a cookie?

A cookie is used as client-side storage for information - they're mostly used in order to store certain user preferences, authentication details (such as the username and password) and session information. In fact cookies can be seen as client-side implementation of sessions and are usually combined with strong cryptographic mechanisms in order to provide safety, integrity and authenticity of the data.

If you are looking for additional material on cookies, RFC 2965 - *HTTP State Management Mechanisms* - from 2000[1] states that cookies are not to be used for account

1 *http://tools.ietf.org/html/rfc2965*

information unless encryption is being used. Section 8 of the updated RFC 6265 from 2011[2] (this RFC supersedes RFC 2965) defines various considerations that must be taken into account to ensure security.

What's a session?

While cookies are being used as client-side mechanism, sessions are stored server-side and serve as a way to persist user interaction with web applications.

Session identifiers are used in order to handle a minimum of details on the client-side but also expose the application to the possibility of session fixation attacks (as outlined in RFC 6265[3]).

How Express handles sessions

Since version 4 of Express a lot of bundled middleware like `bodyParser`, `cookieParser` and `session` got moved into separate modules in order to allow for quicker updates and better maintainability. In this section we will explore some of the functionality that the module `express-session` brings to the table in order to allow for secure handling of sessions.

When using the Express generator or looking at most tutorials, you will see a default initialization of the `session` middleware that looks somehow like this:

```
var session = require('express-session');

app.use(session({
  secret: 'mechagodzilla',
  resave: false,
  saveUninitialized: true,
  cookie: {
    secure: true
  }
}));
```

SHA-256 is part of the family of SHA-2 (Secure Hash Algorithm) hash functions and produces digests (hash values) that are 256 bits by using 32 bit words. The SHA-2 finds usage in popular protocols such as SSL or PGP but got heavy critics in recent events for being used as cryptographic tool in order to secure passwords. Please refer Chapter 2 for additional information about SHA-2 and proper alternatives.

2 *http://tools.ietf.org/html/rfc6265#section-8*

3 *http://tools.ietf.org/html/rfc6265#section-8.4*

Let's look at the default options that are being passed towards the middleware:

- `secret` is a required option and resembles the session secret that is utilized to sign the session ID cookie. The session module relies on the node module `cookie-signature` which utilizes `crypto`, a popular module for cryptography, in order to sign the provided value with `SHA-256` as keyed-hash message authentication code - also known as HMAC. Finally, the signed value is being hashed as `Base64` before being returned to Express.
- The next option, `resave`, decides if sessions are saved in the session storage even when they did not get modified. The default value used to default to `true` but is not encouraged to use anymore. Instead decide based on your session storage: if `req.session.touch()` is used, `resave` can be set to `false`- otherwise `true` is the recommended value[4].
- `saveUninitialized` is used to determine if a new (but unmodified) session should be saved - while `true` is the default value, it is recommend to set this value to `false` when being used to handle authentication.
- `cookie` allows for deeper configuration of your session ID cookie. `secure` is recommended to be set to `true` and ensures that secure cookies for HTTPS-enabled website work. The session middleware does not rely on the `cookieParser` middleware anymore and can even cause issues when being used in conjunction.
- `cookie.maxAge` is set to `null` per default and results in cookies being handled as browser-session cookies - which means that they get removed as soon as the browser window is closed.

Testing the session ID generation can be a difficult task when using the `secure` option for the cookie. The value `true` only serves the cookie via HTTPS. When testing on both HTTP and HTTPS connection the value `auto` can be used - please be aware that a cookie that got set via HTTPS is only accessible for HTTPS connections. Non-secure cookies can be accessed using either HTTP or HTTPS[5].

In our examples we stick to `true` as the default value for our session cookie - this is the recommended setting [6]. When developing in an non-HTTPS-enabled environment consider setting `secure` to `false`.

4 *http://github.com/expressjs/session#resave*

5 *http://www.w3.org/Protocols/rfc2109/rfc2109*

6 *http://github.com/expressjs/session#cookie-options*

A prime feature of this middleware is the possibility of setting up your own function for generating session IDs. `genid` can be used with any function that creates and returns unique IDs. Currently, the default implementation uses `uid2`.

Using genid

While `uid2` seems to be a reasonable implementation of creating session IDs, `genid` provides a way to create harder-to-guess IDs, provide Access Tokens or whichever other scenario you are looking for. Asynchronous functions are sadly not supported as of now but are a listed issue in the project's repository[7].

Let's use `genid` in order to produce our own session IDs:

```
var session = require('express-session');
var uuid = require('node-uuid');

app.use(session({
  genid: function(req) {
    return uuid.v4()
  },
  secret: 'mechagodzilla',
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: true
  }
}));
```

In the above example we generate UUIDs with `node-uuid`. In the following section we will have an in-depth look into how we apply this in practice.

Best practices when dealing with sessions

Sessions are designed to have a finite lifespan and are supposed to expire by either being invalidated through the web application or simple cookie mechanics. Within our application we can tie the generated session towards a timespan and therefore easily validate if the session should remain valid, is invalid or should be invalidated. In order to minimize expensive database operations, it is reasonable to provide the sessions creation date as additional suffix in the session hash itself. This does not only help to reduce computing time but also adds another factor to the generated hash and therefore acts as additional security mechanism. A potential attacker deals not only with the session's id but also with a timestamp. To make session-guessing even harder, we can add additional factors into generating session hashes.

7 *http://github.com/expressjs/session/issues/107*

When using your own function to generate session IDs, you need to be aware about the fact that the session secret provided to Express is not being used (even though it is a required parameter). The module `cookie-signature` is very easy to use and allows for signing and un-signing of cookies:

```
var cookie = require('cookie-signature');

var value = 'test';
var secret = 'myamazingsecret';

var signedValue = cookie.sign(value, secret);
// signedValue is 'test.6L58yh6xptQIl6IyKA5GxGr63TRJwwxTNUYy6ui51Bk'

var unsignedValue = cookie.unsign(signedValue, secret);
// unsignedValue is 'test'
```

Let's apply these best practices to the session generation function we've worked with before:

```
var session = require('express-session');
var uuid = require('node-uuid');

app.use(session({
  genid: function (req) {
    var timestamp = Date.now();
    var id = uuid.v4();
    var sessionId = id + '$' + timestamp;
    return sessionId;
  },
  secret: 'mechagodzilla',
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: true
  }
}));
```

In this example the `$`-symbol is being used as a delimiter. By leveraging this syntax we can easily identify the timestamp after we've retrieved the session's ID. Using this mechanism we do not only benefit from making session-guessing harder; we also have the power of being able to check if sessions are expired already by simply validating the timestamp:

```
var sessionParts = req.sessionID.split('$');
if (sessionParts.length === 2) {
  var timestamp = sessionParts[1];
  // Validate session
  ...
}
```

In this section we've learned how the `express-session` module works, which methods and modules it uses and how we can build our own secure implementation of session ID generation in order to achieve security-layering and make session-guessing harder than before.

Handling XSS

Cross-site scripting is a popular attack that has been briefly introduced in Chapter 4. XSS attacks are based on the fact, that the browser trusts sites it visits, and therefore can be led towards executing malicious code. By injecting this code into other sites it's distributed and either persisted or used for one-off attacks such as filling out forms within web applications.

The Three Types of XSS Attacks

When dealing with XSS prevention mechanisms, three different attack methods become apparent:

1. Persistent XSS
2. Reflected XSS
3. DOM-based XSS

Persistent XSS relies on malicious code being stored in a website's database. This results in the injected code being loaded over and over again and was a popular exploit in the early days of internet forums.

Reflected XSS attacks originate from the original request and can be seen as one-time attacks that are returned in the server's response to the victim.

The last type, DOM-based XSS, is based on modifying client-side code and often attacks the URL's fragment identifier[8] which we know as the hash mark that normally serves for navigational purposes. Another example for DOM-based XSS is modifying entries in HTML5's LocalStorage - OWASP released a cheat sheet when dealing with HTML5 in order to assist with securing your application[9].

Testing XSS Protection Mechanisms

In this section we will explore how a basic reflected injection attack works and which mechanisms Node, HTTP and the browsers put into place in order to prevent exploitation.

8 *http://en.wikipedia.org/wiki/Fragment_identifier*

9 *http://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Local_Storage*

First of all we will generate a new project (using the Express generator) which will be configured in order to be vulnerable. To get started execute the following command in your terminal:

```
express xss
cd express
npm install .
```

This will generate the initial setup for our application. After all modules get installed, we can start working on a very basic user registration feature - residing in `views/index.jade` - which accepts a first- and lastname and eventually displays these details on a simple profile page.

```
extends layout

block content
  h1 Registration
  form(method='POST', action='/register')
    fieldset
      legend Your details
      label(for='firstname') Firstname
      input(id='firstname', name='firstname', type='text')
      label(for='lastname') Lastname
      input(id='lastname', name='lastname', type='text')
      input(type='submit', value='Submit')
```

The form submits a POST requests with the populated body to the route `/register`. In this step we will add this new route declaration to our application (which you can find in `routes/index.js`):

```
router.post('/register', function(req, res) {
  var user = {
    firstname: req.body.firstname,
    lastname: req.body.lastname
  };

  res.render('profile', { user: user });
});
```

For this example we don't bother with storing user data and simply render another template, `profile`, with the POST request's details. The profile page is created by running the following command:

```
touch views/profile.jade
```

The template below displays the user's firstname and lastname using Jade's buffered code mechanism:

```
extends layout

block content
  h1 Your Profile
```

```
    p= user.firstname
    p= user.lastname
```

Now that all pieces are in place, we can run our first attempt at an XSS attack. Instead of submitting an innocent combination of first- & lastname, we want to abuse the lastname in order to submit a script that loads another site:

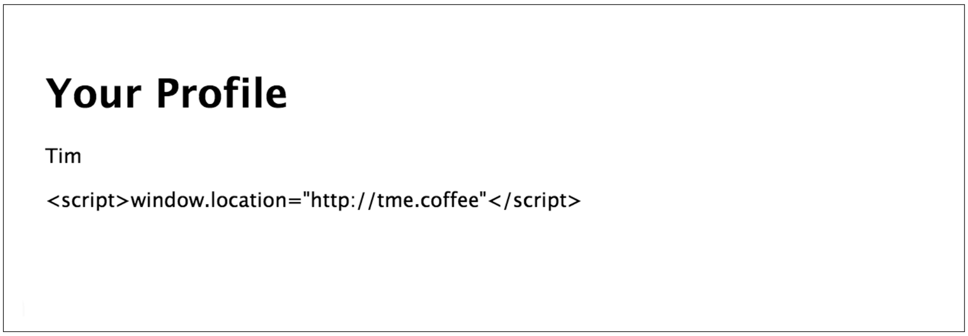
```
<script>window.location="http://tme.coffee"</script>
```

Submitting the details will populate the POST request's body like this:

```
{
  firstname: 'Tim',
  lastname: '<script>window.location="http://tme.coffee"</script>'
}
```

Now the basic assumption is that our profile page would simply render the information provided and therefore loads the URL `http://tme.coffee` instead of displaying the lastname.

Gladly Jade escapes buffered code and therefore simply renders the following output:



Your Profile

Tim

<script>window.location="http://tme.coffee"</script>

Figure 6-2. Escaped Profile Information

When inspecting the page's source, you will notice that the escaped profile page will render like this:

```
<body>
  <h1>Your Profile</h1>
  <p>Tim</p>
  <p>&lt;script&gt;window.location=&quot;http://tme.coffee&quot;&lt;/script&gt;</p>
</body>
```

For this upcoming step, let's be foolish and disable this automatic escaping-mechanism - this can be done by simply changing the template:

```
extends layout

block content
  h1 Your Profile
```

```
p= user.firstname
p!= user.lastname
```

Instead of displaying buffered code with =, we use the functionality to render unbuffered code != - which is definitely not safe for input - and resubmit our form.

This is where things get interesting as differences in browsers become quite apparent. Chrome (tested with Version 48.0.2564.48 beta) simply doesn't render the lastname's field and prints the following statement in the JavaScript console:

> The XSS Auditor refused to execute a script in *http://localhost:3000/register* because its source code was found within the request. The auditor was enabled as the server sent neither an *X-XSS-Protection* nor *Content-Security-Policy* header.

Doing the same experiment with Mozilla's Firefox V42 results in the script tag executing - the browser loads up the page `http://tme.coffee`.

XSS Auditor is a feature that initially got introduced with WebKit and made it's way into the Chromium Project's WebKit fork Blink. That implies that a wide array of browsers such as Chrome, Opera and Safari come with this inbuilt mechanism against reflected XSS attacks. Microsoft's Internet Explorer supports XSS Auditor functionality with version 8 and above.

Circumventing XSS Auditor

XSS Auditor should not be seen as a sole defense mechanism against XSS attacks. It is rather an additional client-side tool to decrease the risk implied by vulnerable software. By blacklisting certain payloads and query parameters, injection points are supposed to be detected and execution on the browser DOM is prevented by transforming the response into a non-executable state.

Let's have a look at the script tag detection mechanism in WebKit's XSSAuditor.cpp[10]:

```
static bool startsOpeningScriptTagAt(const String& string, size_t start)
{
    return start + 6 < string.length() && string[start] == '<'
        && WTF::toASCIILowerUnchecked(string[start + 1]) == 's'
        && WTF::toASCIILowerUnchecked(string[start + 2]) == 'c'
        && WTF::toASCIILowerUnchecked(string[start + 3]) == 'r'
        && WTF::toASCIILowerUnchecked(string[start + 4]) == 'i'
        && WTF::toASCIILowerUnchecked(string[start + 5]) == 'p'
        && WTF::toASCIILowerUnchecked(string[start + 6]) == 't';
}
```

10 *https://github.com/WebKit/webkit/blob/fa65954cef5bc7a64f34e38d08f21833cad81506/Source/WebCore/html/parser/XSSAuditor.cpp#L102*

Based on this detection mechanism potential threads can be detected and examined more deeply.

The auditor relies on the following HTTP header to be set: `X-XSS-Protection`. We can inspect our request's header in order to understand how Express, Node and Chrome handle this situation:

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding:gzip, deflate
Accept-Language:en-US,en;q=0.8,de;q=0.6
Cache-Control:max-age=0
Connection:keep-alive
Content-Length:73
Content-Type:application/x-www-form-urlencoded
Host:localhost:3000
If-None-Match:W/"be-JIAfZIOVAe1p85FawKqWIg"
Origin:http://localhost:3000
Referer:http://localhost:3000/
Upgrade-Insecure-Requests:1
User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko)
```

The sample application does not provide the `X-XSS-Protection` itself - gladly WebKit sets this option as default and it needs to be explicitly disabled by setting the header manually:

```
router.post('/register', function(req, res) {
  var user = {
    firstname: req.body.firstname,
    lastname: req.body.lastname
  };

  res.set('X-XSS-Protection','0');
  res.render('profile', { user: user });
});
```

The header options value `0` disables this mechanism, `1` enables it and `1;mode=block` results in rendering a blank page. Obviously it is always desirable to either set the value to `1` or `1;mode=block`. Warning: While other web frameworks - such as Sinatra[11] for Ruby - provide the `X-XSS-Protection` header as default, Express relies on external modules such as Helmet[12].

11 *http://www.sinatrarb.com*

12 *http://www.npmjs.com/package/helmet*

Setting the X-XSS-Protection header causes vulnerabilities on old versions of Internet Explorer[13]. It is recommended to set the header to `0` accordingly. Helmet handles this exception automatically[14].

The Security Consultant Egor Homakov wrote a very interesting post[15] about the mechanism of XSS Auditor.

Conclusion

Through a combination of sensible browser defaults like the `X-XSS-Protection` header and the features of modern templating engines like escaping input, a solid base-layer security is being provided. Previously, we've been exploring the concept of security-layering and it definitely makes sense to hold on to this behavior when dealing with reflected XSS. As developers we should use both client-side and server-side features that help escaping and sanitizing input in order to prevent exploitation of our application.

More information on XSS attacks and different tools that help with reducing attack vectors can be found here[16].

CSRF attacks

Close to being as popular as XSS, cross-site request forgery is used in order to leverage the browsers trust in a user to execute requests on the user's behalf. This can cause site's to execute requests that seemingly come from a valid authorized user and express a huge threat. In fact, CSRF ranks as the eight-biggest harm in OWASP's top ten list of current security threats[17] down from the fifth place in 2010[18]. While the threat might be decreasing in commonness, a place within the top 10 of ongoing vulnerabilities still justifies discussing the issue itself and prevention mechanisms that help Node.js developer to deploy secure applications.

13 *http://hackademix.net/2009/11/21/ies-xss-filter-creates-xss-vulnerabilities*

14 *http://github.com/helmetjs/helmet#xss-filter-xssfilter*

15 *http://homakov.blogspot.de/2013/02/hacking-with-xss-auditor.html*

16 *http://excess-xss.com*

17 *http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2013*

18 *http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2010*

Handling CSRF with csurf

Express offers an optional middleware called `csurf` [19] that can be installed via npm. Effectively the module provides a unique token that needs to be rendered in forms and will be validated after form-submission - a mechanism that's similar to what we have done with providing the `state` parameter when requesting the Authorization Code in the previous sample.

```
var csurf = require('csurf');

var csrfMiddleware = csurf({
  cookie: true
});

app.get('/form', csrfMiddleware, function(req, res) {
  res.render('form', { csrfToken: req.csrfToken() });
});
```

In this example we create a new middleware based on `csurf` that stores uses cookies instead of `req.session` in order to store the CSRF token secret. This middleware is mounted for the route `/form` and provides the generated token as option to the `form` template.

```
extends layout

block content
  h1 CSRF protection using csurf
  form(action="/login" method="POST")
    input(type="text", name="username=", value="Username")
    input(type="password", name="password", value="Password")
    input(type="hidden", name="_csrf", value="#{csrfToken}")
    button(type="submit") Submit
```

By rendering the CSRF token as hidden part of a hypothetical login form, we provide a way to send the token back to the server. This allows to validate the authenticity of our request in the `/login` route.

```
app.post('/login', csrfMiddleware, function(req, res) {
  // This request is unique and can be handled accordingly
});
```

Some of the functionality in `csurf` relies on using a session middleware (like `cookie-parser` or `express-session`) before using the middleware itself. This is just a matter of handling the initialization process of your application's middlewares accordingly.

19 *http://github.com/expressjs/csurf*

Protecting our application against CSRF attacks with popular modules such as `csurf` is incredibly easy and leaves little excuse for not doing so. By simply using a middleware and handling the provided token, a whole level of threat can be managed.

Valuable resources for Node

Lusca

Back in 2012 PayPal started adopting Node in order to power its application stack. Part of this process was the development of krakenjs[20], an extension of Express. Part of the kraken suite is a module called Lusca, which focusses heavily on improving security by setting sensible defaults and coming with protection mechanisms against prominent vulnerabilities.

Lusca can be used as application-level middleware by adding it to your application's initialization process (sample taken from the project's documentation[21]):

```
var express = require('express'),
    app = express(),
    session = require('express-session'),
    lusca = require('lusca');

//this or other session management will be required
app.use(session({
    secret: 'abc',
    resave: true,
    saveUninitialized: true
}));

app.use(lusca({
    csrf: true,
    csp: { /* ... */},
    xframe: 'SAMEORIGIN',
    p3p: 'ABCDEF',
    hsts: {maxAge: 31536000, includeSubDomains: true, preload: true},
    xssProtection: true
}));
```

Please note that not only common threats such as XSS and CSRF are covered, Lusca also features HTTP Strict Transport Security (HSTS), Content Security Policy (CSP) options, and support for the X-Frame-Options response header.

20 *http://krakenjs.com*

21 *https://github.com/krakenjs/lusca#usage*

helmet

Helmet[22] is a collection of 10 security middlewares that operate closely to how PayPal's Lusca functions. You can either use a default initialization with `app.use(helmet())` and cover 6 out of 10 middlewares (as documented in the project's repository[23]) or use the submodules individually.

Currently, Helmet comes with the following modules:

- `contentSecurityPolicy` - CSP settings
- `dnsPrefetchControl` - controls browser DNS prefetching. Currently beta and soon part of the default middleware.
- `frameguard` - prevention of clickjacking
- `hidePoweredBy` - controls the X-Powered-By header
- `hpkp` - HTTP Public Key Pinning
- `hsts` - HTTP Strict Transport Policy
- `ieNoOpen` - X-Download-Options (IE8 and newer)
- `noCache` - client-side caching
- `noSniff` - prevents MIME type sniffing
- `xssFilter` - X-XSS-Protection header

We definitely recommend reading Helmet's documentation - the contributors took the time to provide detailed explanations of the different attack types and how they can be mitigated[24].

Node Security Project

The last valuable resource that is worth mentioning is the Node Security Project. It is definitely worth to subscribe to their Node Security Newsletter and trying out `nsp`. `nsp` allows for auditing your Node application's `package.json` and `npm-shrinkwrap.json` files for known vulnerabilities.

Simply install `nsp` by running the following command:

```
npm install -g nsp
```

Afterwards navigate to your project and run `nsp` like so:

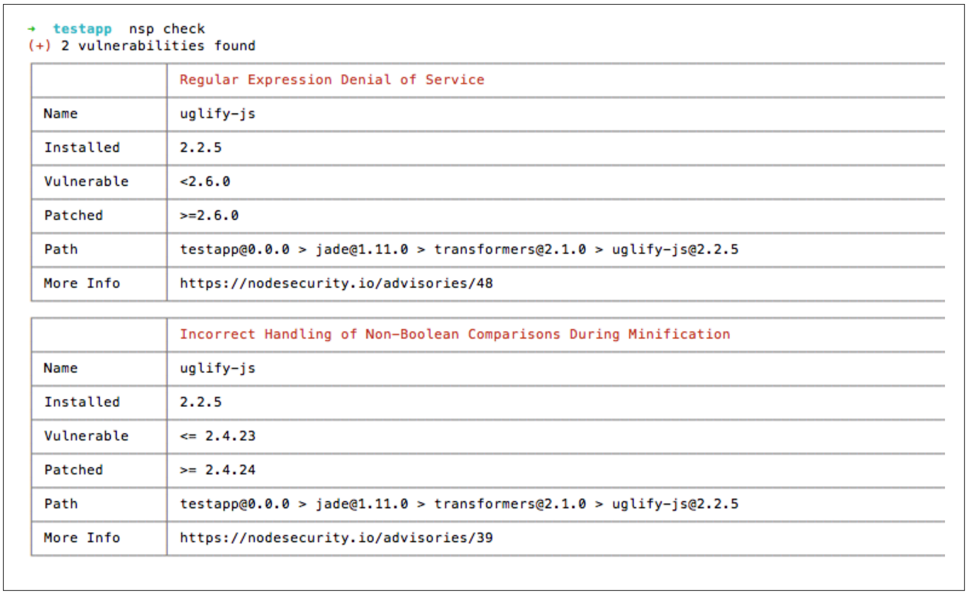
22 *http://github.com/helmetjs/helmet*

23 *http://github.com/helmetjs/helmet#how-it-works*

24 *http://github.com/helmetjs/helmet#usage-guide*

```
cd testapp
nsp check
```

Potential output will look similar to the following:



```
→  testapp  nsp check
(+) 2 vulnerabilities found
┌──────────────┬────────────────────────────────────────────────────────────────────┐
│              │ Regular Expression Denial of Service                               │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Name         │ uglify-js                                                          │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Installed    │ 2.2.5                                                              │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Vulnerable   │ <2.6.0                                                             │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Patched      │ >=2.6.0                                                            │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Path         │ testapp@0.0.0 > jade@1.11.0 > transformers@2.1.0 > uglify-js@2.2.5 │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ More Info    │ https://nodesecurity.io/advisories/48                              │
└──────────────┴────────────────────────────────────────────────────────────────────┘

┌──────────────┬────────────────────────────────────────────────────────────────────┐
│              │ Incorrect Handling of Non-Boolean Comparisons During Minification  │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Name         │ uglify-js                                                          │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Installed    │ 2.2.5                                                              │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Vulnerable   │ <= 2.4.23                                                          │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Patched      │ >= 2.4.24                                                          │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ Path         │ testapp@0.0.0 > jade@1.11.0 > transformers@2.1.0 > uglify-js@2.2.5 │
├──────────────┼────────────────────────────────────────────────────────────────────┤
│ More Info    │ https://nodesecurity.io/advisories/39                              │
└──────────────┴────────────────────────────────────────────────────────────────────┘
```

Figure 6-3. nsp security audit

The Node Security Project also curates a list of valuable resources[25] (such as talks, papers, blog posts and more). This collection serves as a great overview about current security best practices for Node.

Other mitigation techniques

A lot of different mitigation techniques - such as HSTS or CSP - have been mentioned in the above sections. We'd like to take the time to briefly summarize their core functionality before moving on ot the next chapter.

Table 6-1. Mitigation techniques

mechanism	description
Content Security Policy[a]	Prevents execution of malicious content in trusted sites (reduces risk of XSS).
HTTP Strict Transport Security[b]	A mechanism that allows applications to be only accessed via secure connections.
HTTP Public Key Pinning[c]	Ensures the authenticity of a server's public key.

25 *http://nodesecurity.io/resources*

mechanism	description
Frameguard[d]	Allows to define if an application allows to be framed (in a `<frame>` or `<iframe>` tag).
Platform for Privacy Preferences[e]	P3P allows to define which information a website collects about users.

[a] *http://content-security-policy.com*

[b] *http://tools.ietf.org/html/rfc6797*

[c] *https://developer.mozilla.org/en/docs/Web/Security/Public_Key_Pinning*

[d] *https://en.wikipedia.org/wiki/Clickjacking*

[e] *http://www.w3.org/P3P/*

The `X-Powered-By` header does not necessarily open up a vulnerability but tells potential attackers about our application stack. By default Express sets the header to `X-Powered-By: Express` and therefore exposes which framework got used to build the application. We recommend to unset the header either manually (like outlined below) or to use the options provided by Helmet and Lusca.

```
router.get('/myroute', function(req, res) {
  res.set('X-Powered-By', '');
});
```

We highly recommend to do some further research on these mititgation techniques in order to understand how to implement them most effectively.

Our findings

In this chapter we have intensively discussed and dissected session mechanisms, the potential threat behind cross-site scripting and the severe impact of cross-site request forgery. So far we have learned about:

- Express security features (such as `express-session`)
- Client-side XSS prevention (XSS Auditor and the `X-XSS-Protection` header)
- Server-side XSS prevention (escaping and buffering parameters using templating engines)
- Securing requests against CSRF (providing unique tokens with `csurf`)
- Using security modules (`helmet` and `Lusca`)
- Mitigation techniques (CSP, HSTS, HPKP, and more)

In the following chapter we will have a look at efficient and secure data transmission using techniques such as SSL, asymmetric and symmetric encryption and the implications of cryptography itself.

CHAPTER 7

Data Transmission Security

Jonathan LeBlanc

In Chapter 2 we discussed at length about protection of identification and account security through the use of proper hashing and salting techniques. Even though account security is vitally important to any system, what if we want to secure any data that is being transmitted from one party to another, where that data might be sensitive in nature or contain priviledged user information.

There are numerous techniques towards data security that we will explore, which are designed to protect data in motion, or better said, data that is moving between parties. Throughout this chapter we'll look at a few of these techniques in depth:

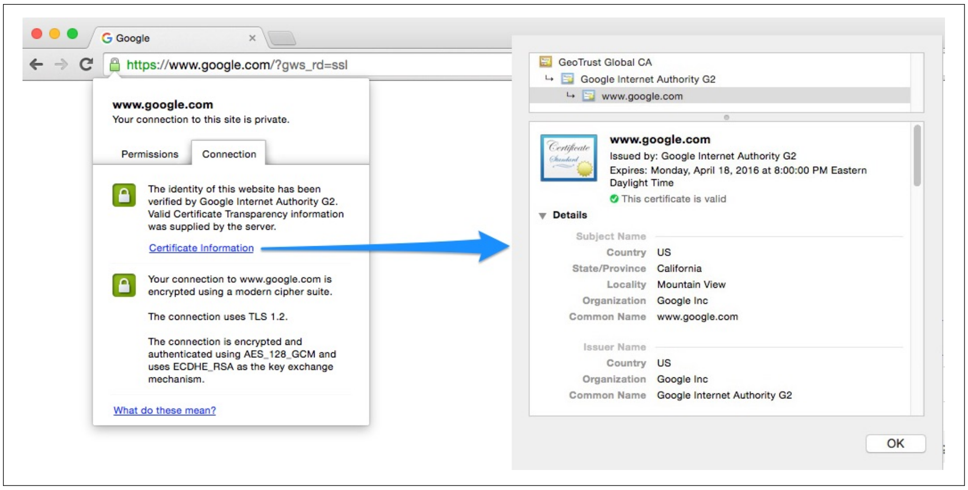
- The concepts behind SSL secure data transmission.
- Asymmetric key cryptography, better known as public / private key encryption.
- Symmetric key encryption, better known as shared secret encryption.

Let's start things out by exploring our ideal secure scenario.

SSL/TLS

Under the ideal scenario, when working with data security as web developers, Secure Sockets Layer (SSL) is the mechanism that you should be targeting as your data security standard for a user.

If you're not familiar with how it works, you'll be familiar with seeing the affect of an SSL certificate being used on websites that you visit, such as the below image, which shows the expanded certificate information for *https://www.google.com/*.


Google

https://www.google.com/?gws_rd=ssl

www.google.com
Your connection to this site is private.

Permissions | Connection

The identity of this website has been verified by Google Internet Authority G2. Valid Certificate Transparency information was supplied by the server.

Certificate Information

Your connection to www.google.com is encrypted using a modern cipher suite.

The connection uses TLS 1.2.

The connection is encrypted and authenticated using AES_128_GCM and uses ECDHE_RSA as the key exchange mechanism.

What do these mean?

GeoTrust Global CA
↳ Google Internet Authority G2
↳ www.google.com

www.google.com
Issued by: Google Internet Authority G2
Expires: Monday, April 18, 2016 at 8:00:00 PM Eastern Daylight Time
This certificate is valid

Details

Subject Name	
Country	US
State/Province	California
Locality	Mountain View
Organization	Google Inc
Common Name	www.google.com
Issuer Name	
Country	US
Organization	Google Inc
Common Name	Google Internet Authority G2

OK

Figure 7-1. SSL Certificate on Google

SSL, and its successor, Transport Layer Security (TLS), are cryptographic protocols that are typically bundled together when talking about "SSL". When creating a secure SSL connection on your website, you will be required to set up an SSL certificate, which is provided by an SSL certificate authority (CA), which is the company that can issue these digital certificates. When doing so, you verify identification information about your site to the certificate authority, then your web server generates two cryptographic keys, a public and a private key. This process uses the symmetric key cryptography approach to data security and data privacy, which we'll be exploring in more detail later in this chapter.

These certificates usually contain some basic information about you and your website:

- Domain name
- Company name
- Address
- City
- State / Province
- Country

Certificate Validation Types and Authorities

There are a number of different SSL certificate types that you can obtain when working through a certificate authority, depending on your needs. These include:

Domain Validation (DV): The CA validates that the applicant has the right to use the specific domain name, meaning that someone with admin rights to the domain is aware of the application. Rights are typically proven by either receiving and confirming an email that is sent to the admin email for the domain, or by configuring specific DNS records for the domain. No company information is vetted or displayed to customers visiting the website who view the certificate details.

With domain validation, you will see the green lock in the URL bar, but will not see company specific details when the certificate is loaded.

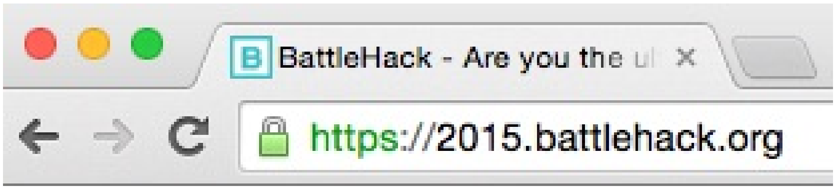
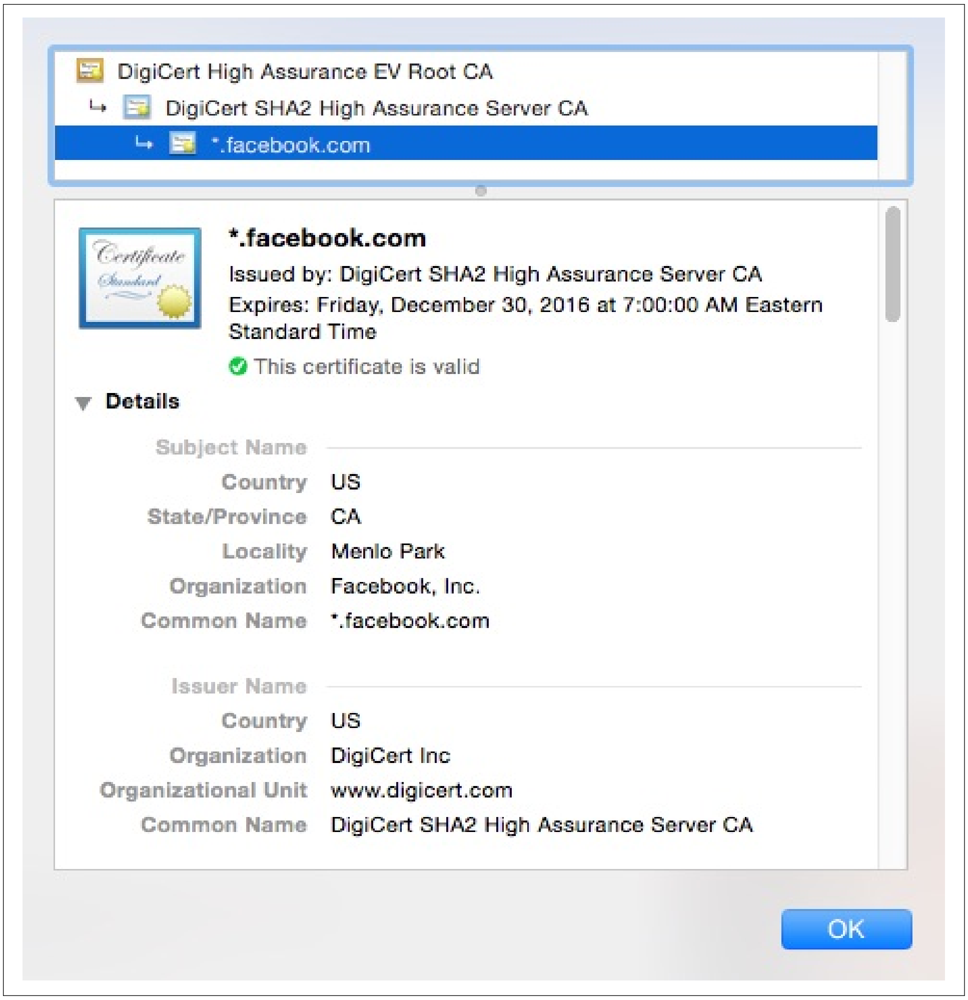


Figure 7-2. Domain Validation Certificate Example

Organization Validation (OV): The domain verification in the DV step is conducted, but in addition, the company / organization information goes through some vetting, such as name, city, state and country. This basic information will also be displayed to customers visiting the website who view the certificate details.

With sites that have this type of validation, you will be able to see company information when the certificate is loaded, like below:


DigiCert High Assurance EV Root CA
↳ DigiCert SHA2 High Assurance Server CA
↳ *.facebook.com

*.facebook.com
Issued by: DigiCert SHA2 High Assurance Server CA
Expires: Friday, December 30, 2016 at 7:00:00 AM Eastern Standard Time
This certificate is valid

Details

Subject Name	
Country	US
State/Province	CA
Locality	Menlo Park
Organization	Facebook, Inc.
Common Name	*.facebook.com
Issuer Name	
Country	US
Organization	DigiCert Inc
Organizational Unit	www.digicert.com
Common Name	DigiCert SHA2 High Assurance Server CA

Figure 7-3. Organization Validation Certificate Example

Extended Validation (EV): The CA validates domain ownership (DA), organization information (OV), as well as checks the legal existance of the organization. This is the lengthiest process of the three, and also validates that the organization is aware of the SSL certificate request and approves it. The validation step requires specific documentation that certifies the identity of the company, as well as an additional set of steps and checks.

Typically you will see an example of this type of validation in sites that have a green address bar with the lock and company name, as displayed below.

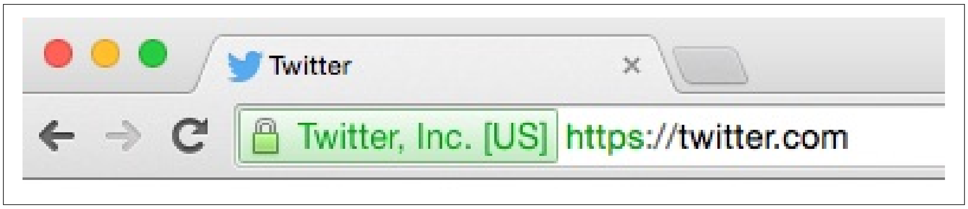


Figure 7-4. Extended Validation Certificate Example

When working with a CA, such as Digicert, GoDaddy, Verisign, Comodo, etc, you will typically be brought through a few steps to:

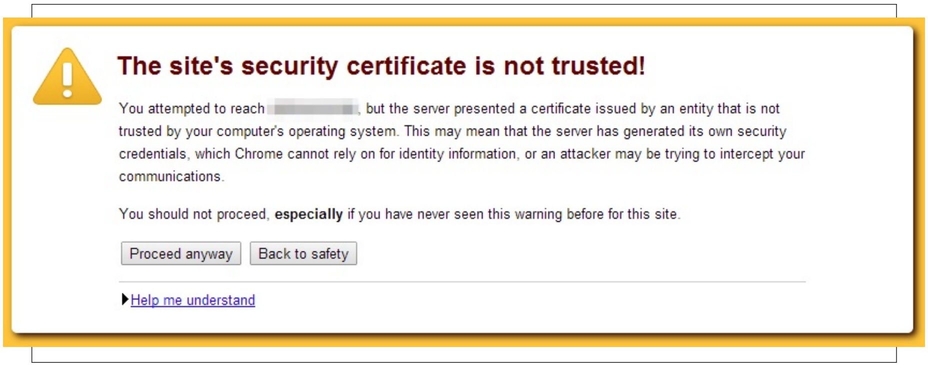
- Create a Certificate Signing Request (CSR)
- Purchase the certificate

Once the certificate is purchased, the CA will validate and process the CSR, then issue you the certificate for installation (typically through email). Many CAs offer support for Wilcard certificates, which allows you to also secure all subdomains under a root domain.

Creating your own Self-Signed Certificate for Testing

Using in production will produce an error

Working with self-signed certificates, as we will be seeing in this section, should never be done in a production environment where you are expecting actual traffic. The entire purpose for working with self-signed certificates is for testing only. Using them will produce a nasty browser warning explaining to all your visitors that your certificate is not trusted. You have been warned. The message looks like the below, which will take over the entire browser window before going on to your site content, as displayed below:



The site's security certificate is not trusted!

You attempted to reach [redacted], but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Figure 7-5. Untrusted Certificate Warning

Now that we have been thoroughly warned about using the following section for testing only, let's proceed to learn about creating and signing our own certificates for testing purposes. This will allow us to build our infrastructure in a separate environment before moving to a proper production environment with a trusted certificate authority.

In the previous sections we learned about the fundamentals of setting up a production level certificate. Now let's go through the steps that will need needed, in contrast, for setting up your own certificate.

Certificate Setup

The first step is to create your private key and the self-signed certificate that we will be using when we spin up a Node server instance.

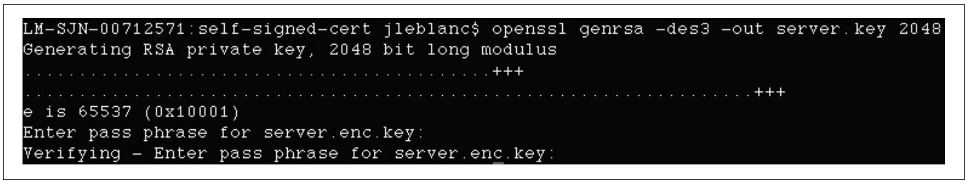
For the sake of the example, we're going to store our keys in the same folder as the program that we will be running. With that said, load up a terminal window and go to the folder you are using for this program.

Let's start with the private key setup. Type in the following command:

```
openssl genrsa -des3 -out server.key 2048
```

When run, you will be asked to enter and verify a password for the file. What you are essentially doing is creating a new 2048-bit, triple DES encrypted, RSA key, encrypting it with a password, and then storing it to a file, *server.enc.key*.

The process will look like the below:



```
LM-SJN-00712571:self-signed-cert jleblanc$ openssl genrsa -des3 -out server.key 2048
Generating RSA private key, 2048 bit long modulus
..........................................+++
.................................................................+++
e is 65537 (0x10001)
Enter pass phrase for server.enc.key:
Verifying - Enter pass phrase for server.enc.key:
```

Figure 7-6. Generating a 2048 bit RSA Key

Next up, we need to create a certificate signing request (CSR). Using our previously generated key, we issue the following command:

```
openssl req -new -key server.key -out server.csr
```

This will create our certificate signing request, with the intent of outputting the CSR to *server.csr*. After issuing the command, this is where you will be required to input more detailed information about yourself and your company. During this process you will be asked to enter:

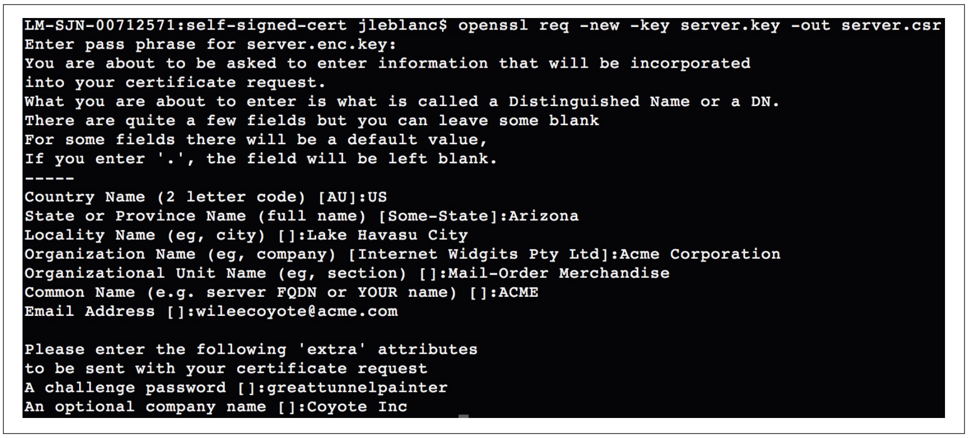
- The passphrase used for the private key (the same one used for the last command).

- Your 2-digit country code.
- Your state or province.
- Your city or locality name.
- Your company or organization name.
- The group or unit that this belongs to in the company.
- Any common name to be used for the certificate request, or your name.
- An appropriate email address to contact.

You will also be asked to add a few extra attributes, including:

- A challenge password.
- A company name (optional).

The entire process, end-to-end, will look like the below:



```
LM-SJN-00712571:self-signed-cert jleblanc$ openssl req -new -key server.key -out server.csr
Enter pass phrase for server.enc.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Arizona
Locality Name (eg, city) []:Lake Havasu City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Acme Corporation
Organizational Unit Name (eg, section) []:Mail-Order Merchandise
Common Name (e.g. server FQDN or YOUR name) []:ACME
Email Address []:wileecoyote@acme.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:greattunnelpainter
An optional company name []:Coyote Inc
```

Figure 7-7. Creating a Certificate Signing Request

Removing Key Encryption

If you are only using the self-signed certificate for testing and wish to remove the password and encryption from your private key (not recommended for anything but single user, local testing), then at this point you could use your previously generated encrypted private key (say that's server.enc.key), and generate the new unencrypted private key (server.key) with the following command:

```
openssl rsa -in server.enc.key -out server.key
```

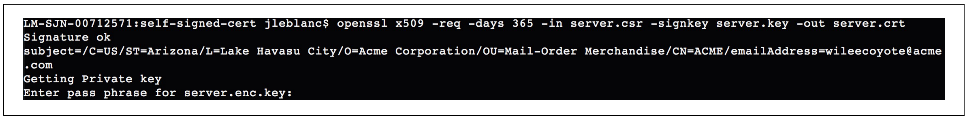
The list of commands to issue to get to the same point as we were in this section, using this method, are:

```
openssl genrsa -des3 -out server.enc.key 2048
openssl req -new -key server.enc.key -out server.csr
openssl rsa -in server.enc.key -out server.key
```

Now that we have our CSR in place, we can self-sign the certificate to create our needed certificate file (CRT), using the following command

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

This command will create a temporary certificate for 1 year (specified by the *-days* option). If you did not remove the encryption from the original key file, then you will be required to enter the key passphrase for this step, as shown below:



```
LM-SJN-00712571:self-signed-cert jleblanc$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
Signature ok
subject=/C=US/ST=Arizona/L=Lake Havasu City/O=Acme Corporation/OU=Mail-Order Merchandise/CN=ACME/emailAddress=wileecoyote@acme
.com
Getting Private key
Enter pass phrase for server.enc.key:
```

Figure 7-8. Creating the Self-Signed Certificate

Our self-signed certificate is now ready for us to use in our server setup, to start creating secure HTTPS connections between the server and browser.

Server Setup

The server code for the below example is available in its entirety at *https://github.com/iddatasecuritybook/chapter7/blob/master/self-signed-cert/server.js*

For the sake of our example server, we're going to assume that you chose not to remove the encryption from your private key in the last section, but we'll still go through how to adjust the code below in case you no longer need the passphrase. We're also assuming that you have Express setup as we go through the code sample below.

We're going to look at an Express server setup that will use our private key and self-signed certificate from the last step to accept secure (HTTPS) connections from an alternate source, such as the browser or another program that we're running locally. In this case, we will be working with JSON strings that will be sent from another program to our server, which will be done through a secure communication channel thanks to our certificate and key.

We will be working with a few modules for our server, the file system (fs), https, and querystring, which are all standard modules and don't require pulling anything from npm, as well as body-parser, which will allow us to support JSON / URL encoded bodies in Express 4.0 or beyond. We install body-parser from npm with the following command:

```
npm install body-parser --save
```

Let's look at the server code in its entirety, then describe what's going on in each step.

```
var fs = require('fs'),
    https = require('https'),
    querystring = require('querystring'),
    bodyParser = require('body-parser')
    app = require('express')();

//support JSON & URL encoded bodies
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));

//handle all POST requests
app.post('/', function (req, res){
    var message = req.body;
    res.send('Message received:' + querystring.stringify(message));
});

//set certificate options
var options = {
   key: fs.readFileSync('server.key'),
   cert: fs.readFileSync('server.crt'),
   passphrase: 'YOUR KEY PASSWORD'
};

//create server with certificate options
https.createServer(options, app).listen(3000, function () {
   console.log('Server started: Listening on port 3000');
});
```

In the code sample above, you can see that we refer to a passphrase, and your key password as the input. These values should never be hard coded in your code. Instead, you should use environment variables, files with restricted read permissions, or the like. The Node module dotenv is a great way to handle these environment variables. A complete runthrough of its use is available in Appendix B, within the *Application Configuration* section.

We first start off by including all of our required modiles, then adding in our body-parser options to Express for supporting JSON and URL encoded strings. Those sections comprise our first two blocks of code above.

Next we need to handle all incoming POST requests that will be coming to the server. We do that by setting **app.post(/, ...)**. In the return function, we first start by extracting **req.body**, which will be the POST object that the browser / other program sent over. In this sample, we then simply send back an acknowledgement that the message was received.

In the next block we set up our certificate options to create that secure connection. Within the **options** variable, we add three pieces of data:

- **key**: Our private key, read in from our local server.key file.
- **cert**: Our self-signed certificate, read in from our local server.crt file.
- **passphrase**: The passphrase for our private key file. If you removed the encryption from this key file in the previous section, you can omit this line.

Lastly, we simply create our server instance, with those options, and listen on port 3000. When you're ready to test this out, you can issue the following command on your terminal, assuming the file is saved as *server.js*:

```
node server.js
```

The server will spin up on port 3000 and will start listening for incoming traffic. Our next step is to set up another script to act as our client, and send it over an appropriate JSON object over our new HTTPS connection.

Making Secure Requests to Server

The client code for the below example is available in its entirety at *https://github.com/iddatasecuritybook/chapter7/blob/master/self-signed-cert/client.js*

Our client script, *client.js* is mostly just a fairly standard POST request, with a few differences. We'll look at the full code below, then describe each of the sections (and any caveats).

```
var querystring = require('querystring'),
    https = require('https');

//POST data to be sent to server
var postData = querystring.stringify({
    'message' : 'My secure JSON string'
});

//POST options
var postOptions = {
    hostname: 'localhost',
    port: 3000,
    path: '/',
    rejectUnauthorized: false,
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Content-Length': postData.length
    }
};

//set up HTTPS POST request to server
var postReq = https.request(postOptions, function(res){
    res.setEncoding('utf8');
    res.on('data', function (data){
        console.log(data);
    });
});

//POST data to server
postReq.write(postData);
postReq.end();
```

We start things off by including the querystring and https standard Node modules.

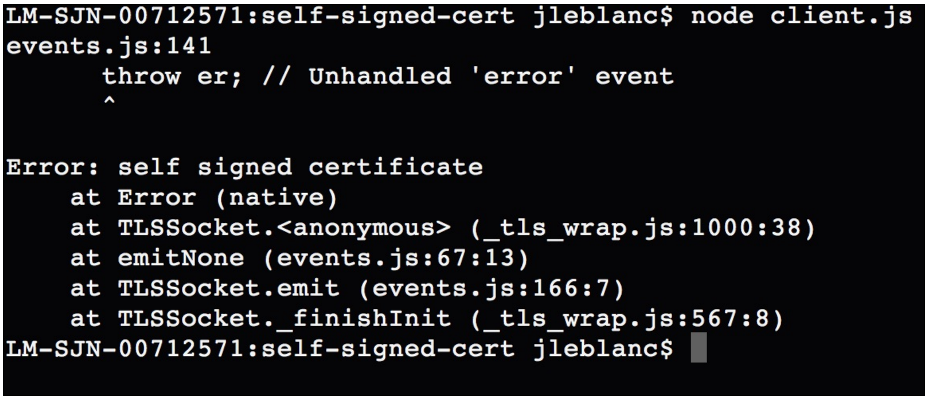
We then create the JSON object that we will be sending through the HTTPS POST request, **post_data**. We add just a simple string for our needs, then stringify the entire object for POSTing.

Under the POST options, this is where we need to pay attention to slight differences from a standard POST request with a non self-signed certificate. Within the POST options, we specify a few options:

- **hostname**: The host to send the request to. Since we're running the server locally, this is localhost.

- **port**: The port to make the request to. The server is on port 3000, so that's what we specify here.
- **path**: The path to make the request to. The server is accepting all POST traffic to the same handle, so this can be anything.
- **rejectUnauthorized**: This is the one to take note of. Specifying this as **false** will allow you to make POST requests with a self-signed certificate without being blocked by a certificate error.
- **method**: The HTTPS request method, in this case POST.
- **headers**: Our content headers, specifying content type and length.

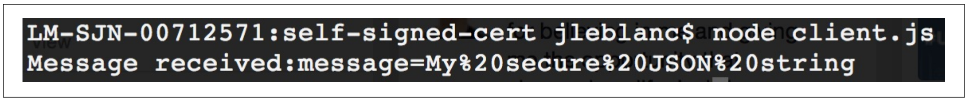
Adding *rejectUnauthorized: false* to your POST options will allow you to work with a self-signed certificate without receiving error messages about the certificate not being trusted. The error produced, when not properly handled, will look like the following:



```
LM-SJN-00712571:self-signed-cert jleblanc$ node client.js
events.js:141
      throw er; // Unhandled 'error' event
      ^

Error: self signed certificate
    at Error (native)
    at TLSSocket.<anonymous> (_tls_wrap.js:1000:38)
    at emitNone (events.js:67:13)
    at TLSSocket.emit (events.js:166:7)
    at TLSSocket._finishInit (_tls_wrap.js:567:8)
LM-SJN-00712571:self-signed-cert jleblanc$ 
```

Next we set up the HTTPS POST request object. We do this by using **https.request(…)**, passing in the **post_options** variable. In the response, we set the encoding of the response, then handling the case where data is sent back from the server with **res.on(…)**. In the case of our client, we are simply logging out the response.

Lastly, we send off the POST request to the server. If all went well, we should see the following response when using our client:



```
LM-SJN-00712571:self-signed-cert jleblanc$ node client.js
Message received:message=My%20secure%20JSON%20string
```

Figure 7-9. Response from HTTPS POST Request

Asyncronous Cryptography

Asyncronous cryptography, also known as public / private key encryption, involves using sets of public / private key pairs for a sender and receiver to encrypt and sign (sender), then decrypt and verify (receiver), any message that should be transmitted securely over a potentially insecure channel, such as when SSL is not available.

Some of these use cases may include multi user environments over internet enabled hardware, small scale microelectronics, or in any case where you might not be able to ensure the integrity of the connection for users.

What we have to know beforehand is how we will be working with these key sets (public / private keys) in order to secure the message being sent. The public key part of the set can be known to anyone needing it (much like a username), while the private key must remain known only to the party it belongs to (much like a password).

Use Case

Let's look at a practical scenario of when this will be valuable. Suppose you're walking around a store with your mobile phone, and the store uses a series of BLE (bluetooth low energy) devices, also known as beacons, for in-aisle purchasing. Basically, the device will allow a phone to connect to it, and then it will in turn connect to some web endpoint to process the message. In this case, the messages may be to check the user in with their account, or to pay for their goods through a store credit card or PayPal account. These BLE beacons themselves are not secure devices, and can be modified by a malicious party. We want to ensure that when the user is sending their login information, the beacon hasn't been modified to now transmit that clear text data to a malicious endpoint, instead of the store endpoint.

This is where public / private keys come in. The person sending the message would encrypt the message with the public key of the recipient (perhaps through a public keystore that verifies requests from the application asking for the public key), and then sign the message with their private key (in the case of the person walking around the store, this might be embedded within their store application). The encrypted and signed versions of the message are transmitted through the beacon, and on to the store API endpoint. That store API endpoint would then decrypt the encrypted message with their private key (perhaps stored securely on their servers), and the signed version would be verified using the public key of the sender (the person walking around the store).

We have now decrypted the message and verified its origins. In doing so, we've mitigated a number of potential attack vectors:

- A malicious party cannot simply watch clear text traffic through the beacon and store any sensitive information going back and forth, because we encrypt the data end-to-end.
- A malicious party cannot send fake data through to the endpoint with hopes of gaining user account access, as the private key is only known to the user, and the message verification would fail without the public key signing.
- A malicious party cannot force the beacon to push data to their own endpoints and expect any valid results. The data might arrive, but without a method of decrypting the data or checking its validity, the information would not be beneficial.

Let's break this down into a simple process, based on the figure below

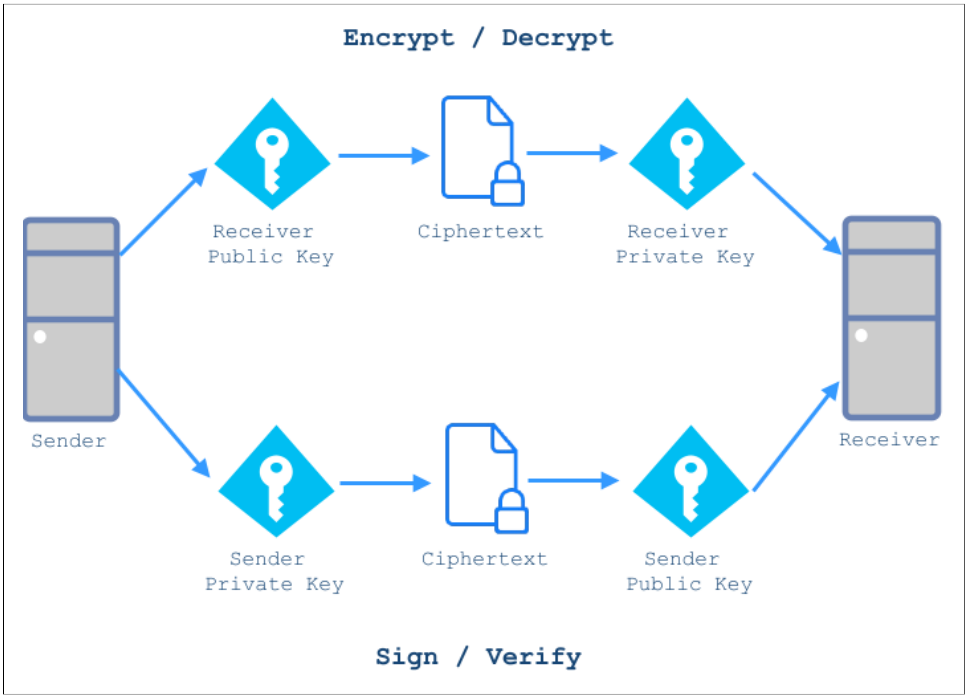


Figure 7-10. Asymmetric Cryptography Process

These are the steps that you would follow as you transmit data:

- You generate two sets of keys, public / private keys for the person sending the data (sender), and another public / private key set for the person receiving the data (receiver).
- You create a small message (typically a string / JSON object or the like) that you would like to transmit.
- From the sender side, you take that message and encrypt it with the receiver's public key, then take the encrypted message and sign it with your private key.
- You transmit the encrypted and signed data to the recipient, sometimes through an intermediate device or service.
- The recipient verifies the signed payload with the public key of the sender, then (if valid) decrypts the encrypted message with their private key.
- We now have a verified and decoded message that can be processed as needed.

Implementation Example

Let's see this functionality in practice through a Node implementation of this process, step by step. We're going to break this up into a 3 step process to explore the main features of the example:

- Steps 1a / 1b: Generating your two sets of public / private keys. 1a covers doing so directly in code, with no storage of the keys, while 1b takes that concept a bit further and shows you how to implement file storage for the public / private keys on top of that. For a proof of concept, 1a is an expedient way to get started, but for all production implementations, 1b is preferred. Choose one of these implementations as you are running through the samples.
- Step 2: Encrypting and signing some piece of data from the side of the sender, to be sent through to the receiver.
- Step 3: Decrypting and verifying the data that was sent from a sender.

When keys should be generated and used

Even though we are showing a full end to end example that can be executed in a single pass, in a production environment this is most likely not the way that the code execution should be structured. If you have a sender / receiver relationship, typically the sender device will be registered with the receiver's service. When that device / site registration occurs, the public / private keys for that pairing (steps 1a or 1b) should be run. Only when the user begins using the service to transmit data (steps 2 and 3) should those keys then be extracted and used.

Let's start out with generating keys.

Step 1a: Generating Keys Without File Storage

The complete sample code for the asymmmetric key cryptography process without using the file system is available at *https://github.com/iddatasecuritybook/chapter7/blob/master/asymmetric-crypto/crypto_no_fs.js*

The first thing we need to do when generating our needed public / private key pairs is to add in a Node package to help us with generating, encrypting, decrypting, signing, and verifying our keys.

There is a popular package, named **ursa**, for doing just that, and we install it like so:

```
npm install fs --save
```

Next, we add that package as a requirement at the top of our Node script.

```
var ursa = require('ursa');
```

Now that we have our packages defined, it's time to generate a few public and private keys that we are going to need during the process. As mentioned early, in this example we are going to simply generate these into variables without any file storage component.

```
//generate sender private and public keys
var senderKey = ursa.generatePrivateKey(1024, 65537);
var senderPrivKey = ursa.createPrivateKey(senderKey.toPrivatePem());
var senderPubKey = ursa.createPublicKey(senderKey.toPublicPem());

//generate recipient private and public keys
var recipientKey = ursa.generatePrivateKey(1024, 65537);
var recipientPrivKey = ursa.createPrivateKey(recipientKey.toPrivatePem());
var recipientPubKey = ursa.createPublicKey(recipientKey.toPublicPem());
```

In this code snippet, we are running the same three lines of code to generate keys for the sender and recipient.

When generating keys, the first item on our list is to make a request to the ursa **generatePrivateKey(…)** method, which will produce a random key set that will be used to extract our individual public and private key pairs.

The parameters passed to **generatePrivateKey()** are:

- The number of bits in the modulus (in the case above, 1024). Anything 1024 or over is generally considered secure, but the method defaults to 2048 if no number is specified.

- The exponent value, which must be odd. This argument is optional as well, and defaults to 65537.

We then need to break those up into our individual matching keys, using the ursa methods **createPrivateKey(…)** and **createPublicKey(…)**, which will accept our previously generated key set in the previous line as a parameter.

That parameter can be passed through as **senderkey.toPrivatePem()** or **senderkey.toPublicPem()**, depending on if you're attempting to generate a public or private key.

Next, let's look at doing the same thing file storage for the keys.

Step 1b: Generating Keys with File Storage

The complete sample code for the asymmmetric key cryptography process using the file system is available at *https://github.com/iddatasecuritybook/chapter7/blob/master/asymmetric-crypto/crypto_fs.js*

Now, let's look at more of a production level deployment, where we might be working with thousands of key pairs stored in a keystore, or having individual sender keys deployed through an application on a user's device.

Like the last example, we'll include the **ursa** package for working with our public / private key pairs, but in addition to those we'll also include a few others, **fs** for generating files and storing information on the file system, **path** for normalizing folder / file paths, and **mkdirp** for generating folder structures with some good duplicate folder handling without producing errors.

The **fs** and **path** packages are part of the core modules, so we don't need to define those. For the other two, we install them from npm like so:

```
npm install ursa --save
npm install mkdirp --save
```

As we have done many times before, we now add those packages as requirements to the top of our Node script.

```
var fs = require('fs');
var ursa = require('ursa');
var path = require('path');
var mkdirp = require('mkdirp');
```

We're going to streamline our key generation a bit, and instead of duplicating the code for generating just two sets of keys, we'll put that all together into a single public / private key generation function that we can call.

```
function makeKeys(rootPath, subPath){
    try {
        mkdirp.sync(path.join(rootPath, subPath));
    } catch (err) {
        console.error(err);
    }

    var key = ursa.generatePrivateKey(1024, 65537);
    var privatePem = key.toPrivatePem();
    var publicPem = key.toPublicPem();

    try {
        fs.writeFileSync(path.join(rootPath, subPath, 'private.pem'), privatePem, 'ascii');
        fs.writeFileSync(path.join(rootPath, subPath, 'public.pem'), publicPem, 'ascii');
    } catch (err) {
        console.error(err);
    }
}
```

Working with the file system

Accessing the file system can be very unreliable. Due to this, it's always a best practice to ensure that you are appropriately capturing and handling errors that are produced. A simplified example of that is displayed in the code above.

Our **makekeys** function will accept two parameters, a root path for where the keys will be stored (rootpath), and a subpath for separating out the public / private keys into individual folders (subpath). In the end, we are looking for a folder structure that looks something like this:

- ./keys/sender (which includes the public / private .pem files for the sender).
- ./keys/receiver (which includes the public / private .pem files for the sender).

We start by creating our necessary folder path with **mkdirp.sync()** for the path supplied. The path package will normalize the root and sub paths together for the folder locations to be created.

Working with mkdirp

Instead of producing errors on attempting to create a duplicate folder structure, mkdirp just continues with the next line of code execution, making it nice to work with in an environment where there is the potential of attempting to create a duplicate folder structure.

Over the next three lines, we use the **ursa** package to generate our key pairs, then extract the public and private keys into individual variables.

Lastly, we use the **fs.writeFileSync(…)** method to create our .pem files to hold the public and private keys. **writeFile(…)** in the example above will accept three parameters:

- The path and file name to write. In this case, it's our root and sub folders, with either private.pem or public.pem as the file name.
- The content to write, which is obtained from our variables holding the public and private keys.
- The content type, in this case ascii.

Now that we have our function in place to create a public and private key where we designate, we can create both the key pairs for the sender and receiver with the following three lines:

```
var rootPath = './keys';
makeKeys(rootPath, 'sender');
makeKeys(rootPath, 'receiver');
```

We should now have the directory structure and four .pem files created. In a more realistic deployment, the content of these .pem files might be stored in a properly secured key storage lookup, or public keys separated into a public keystore that allows the lookup of keys for encryption or verification from verified sources, such as from application locations that were registered with the service.

Key storage file types

There are numerous file extension standards that are used for public / private key storage, including (but definitely not limited to) .pem (can be used for the public key or for the entire public / private chain), .key (for just the private key), .pub (for just the public key), .cert (a .pem file with a different file extension that is recognized by Windows explorer), as well as many others. Choose the one that works best for you. For more discussion on this topic, see this serverfault exchange.

Now that we have our keys in place, we can follow the same type of methodology as we did in step 1a, but this time we're going to extract the contents of the keys from the .pem files we just wrote.

```
var rootPath = './keys';

//generate sender private and public keys
var senderPubKey = ursa.createPrivateKey(
    fs.readFileSync(path.join(rootPath, 'sender', 'private.pem')));
```

```
var senderpubkey = ursa.createPublicKey(
    fs.readFileSync(path.join(rootPath, 'sender', 'public.pem')));

//generate recipient private and public keys
var recipientPrivKey = ursa.createPrivateKey(
    fs.readFileSync(path.join(rootPath, 'receiver', 'private.pem')));
var recipientPubKey = ursa.createPublicKey(
    fs.readFileSync(path.join(rootPath, 'receiver', 'public.pem')));
```

We start with a given root path (same as when we generated the keys) that we should pull the keys from. For each key, we use the **ursa** package to create either a public or a private key. Since it is expecting the key content from the files we wrote, we use the **fs.readFileSync(…)** method to pull the content in, passing along the full path to our .pem files. In a full production deployment, these files would reference back to your particular key store, where the .pem files are stored.

Next up, we're going to see how to use these keys to encrypt and sign a message to be sent.

Step 2: Encrypting and Signing a Message

We're now at the stage where the user (the sender) is ready to send some data through a potentially insecure device or third party, on the way to the receiver. To prepare this data to be sent, we need to encrypt and sign the data using the keys that we just prepared.

```
//prepare JSON message to send
var msg = { 'user':'Nikola Tesla',
            'address':'W 40th St, New York, NY 10018',
            'state':'active' };

msg = JSON.stringify(msg);

//encrypt with recipient public key, and sign with sender private key
var encrypted = recipientPubKey.encrypt(msg, 'utf8', 'base64');
var signed = senderPrivKey.hashAndSign('sha256', encrypted, 'utf8', 'base64');
```

Let's say that the data that we are trying to send is a JSON structure with some priviledged information, the **msg** variable above. To start out, let's go ahead and convert that into a string to be encrypted.

Next we use the recipient's public key (most likely obtained from a public key store) to encrypt the data by calling **recipientPubKey.encrypt(…)**, passing in the string to be encrypted.

After we encrypt, we now need to create a signed version of the encrypted data by using the sender's private key to hash and sign the encrypted string that we just created, which we do by calling **senderPrivKey.hashAndSign(…)**, passing in the hashing algorithm (sha256) and the encrypted string.

We now have two variables, one the encrypted version and the other the signed version. We take those two variables and transmit them through the third party device or service.

Let's say the device working as the transmitter between the sender and the receiver was compromised. Without the private key of the recipient the raw JSON structure can't be extracted. In essence, if the data is sniffed it will be completely useless.

We're now at the stage where the data has transferred through the third party device or service, and has arrived safely at the recipient for decryption and verification.

Step 3: Decrypting and Signing a Message

With the data safely in the hands of the recipient, we need to both extract the data and ensure that the data is coming from a valid source through our matching public / private key pairs.

```
//verify message with sender private key
var bufferedMsg = new Buffer(encrypted);
if (!senderPubKey.hashAndVerify('sha256', bufferedMsg, signed, 'base64')) {
    throw new Error("invalid signature");
} else {
    //decrypt message with recipient private key
    var decryptedMsg = recipientPrivKey.decrypt(encrypted, 'base64', 'utf8');
    console.log('decrypted message verified:', decryptedMsg);
}
```

We start out by needing to verify the data that was sent over, to ensure that the data that was originally signed is from the source that we expect it to be. We create a buffer out of the encrypted message, for comparison. We then use **senderPubKey.hashAndVerify(…)**, passing in the same hashing algorithm we used to sign the data, along with the buffer of our encrypted string, and the signed version of the data that we received from the sender. What is happening here is that we are comparing the Encrypted ciphertext against the signed ciphertext that we received from the sender. If they don't match we have an unverified data object, but if they match then we have valid data and can then move to decrypt the data packet.

Once verified, we now decrypt the data that was transmitted over. If you recall, we originally encrypted the data with the recipient's public key, so now we can only decrypt that data using the matching recipient private key by passing the encrypted message to **recipientPrivKey.decrypt(…)**.

If we print that out, you'll now see the original JSON structure that we started with. Now that we have a verified and decrypted message, we can begin using the data however we need to.

Advantages, Disadvantages, and Uses of Aynchronous Cryptography

Aynchronous, or public key Cryptography has a number of advantages when compared against synchronous cryptography methods. In the same light, we also have to deal with a few drawbacks with this method.

Let's look at a few of the benefits here:

- Key convenience: Since each party (sender and receiver) has their own private key, and that key doesn't need to be transmitted, we don't have to deal with the issue of trying to find a secure method for transmitting that data like you do in a synchronous environment.
- Key non-repudiation: With each party owning their own private key, and that value never being shared, the attack case where we might have some piece of data, with an authenticated server, but potentially have a key that was compromised, is a non-issue. Each party is responsible for the secure storage of their private keys, meaning that this issue doesn't come up.

Now, we are dealing with one pretty big negative when working with this method:

- Speed: Due to the fact that there is a lot of processing that needs to happen during encryption and decryption using this method, it is quite slow when compared to a synchronous model.

With all of this said, asynchronous cryptography is not overly well suited for a single user environment, where you might be encrypting data to be sent between two parties that you own, which is where synchronous cryptography shines. This method works well in a multi-user environment, where the channels between a sender and receiver might not be secure.

Synchronous Cryptography

With Asynchronous cryptography, we've explored the realms of transmitting data through potentially insecure channels. The use of public and private keys in the examples that we looked into are a similar methodology behind technologies such as OAuth and OpenID Connect.

Now that we have that understanding, let's see how we can further protect data transmission between two sources that trust each other. Think of it this way, let's say I'm writing an email through my gmail account to to sent to another gmail account, Google is transmitting data through a secure HTTPS connection, and we can safely assume that the email at the end of the line is from an authentic source.

One of the main uses for synchronous cryptography is in a trusted environment. Instead of using two separate public and private keys, we are instead using a shared secret between two sources. Using a single key can significantly increase the encryption and decryption speed, but at the same time, if the single key is compromised, the resulting damage might be worse.

At a high level, let's look at how this process end to end will work with a single shared secret key.

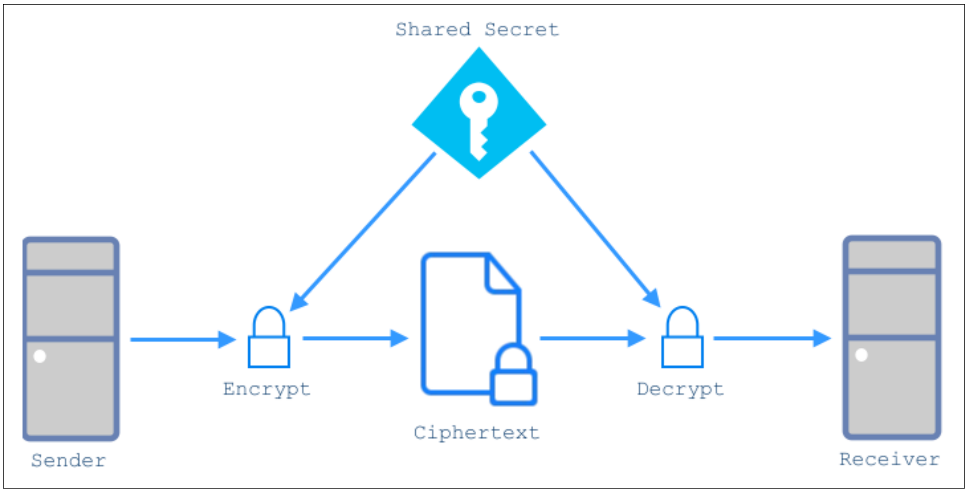


Figure 7-11. Symmetric Cryptography Process

In the above image, we see that:

- The sender has some piece of plaintext data that they intend to send to the receiver over the channel.
- The sender uses the shared secret to encrypt the data, giving us the ciphertext.
- When the receiver obtains the ciphertext, they use the same shared secret to decrypt the ciphertext into its original plaintext.

Below, we'll run through a few practical examples of synchronous cryptography by using AES Block ciphering with a few different modes of operation using an initialization vector. Before we do that though, let's see what initialization vectors and padding are, how these modes of operation differ, and how they affect the block cipher security and performance.

Initialization Vector

Many of the cryptographic algorithms that we are likely to use are what we call "iterative algorithms". When a piece of data that is to be encrypted is split up into blocks for encryption, those iterative algorithms depend on the data from previously encrypted blocks in order to process the next subsequent block. In these cases, the first block of data that will be encrypted has no previous block to build off of, so it needs some piece of data to begin running the algorithm. This is the initialization vector.

An initialization vector is a fixed-sized piece of data that is typically required to be random, or at the least, pseudorandom. Typically, each time you are encrypting a new piece of data, you would supply a new, random, fixed-length piece of data.

Let's look at a practical example of this. In the case of the CBC encryption mode of operation (we will explore that in the next section), the way it works is by breaking up the data to be encrypted into blocks. As the algorithm goes through each block, the plaintext block is XORed ("Enclusive or - A or B but not A and B) against the previous encrypted block, before finally being encrypted together. Since the first block in the chain needs to have a previous block to XOR against, we need to supply that data. That data is the initialization vector.

Padding

In the case of certain block cipher algorithm modes, such as Cipher Block Chaining (CBC) described in the next section, when the data to be encrypted is split up into blocks of data to be processed, the plaintext data that is to be encrypted needs to be an exact multiple of the blocks produced. This means that each block needs to be an equal size. In these cases, if we come across data that cannot be chunked up into equal blocks, we need to add some padding data to the blocks.

There are several standard conventions that are employed that set standards on what this dummy data should be. Let's look at these visually. Let's say that we want to encode the string "This is my block data". If the cipher mode we are employing requires equal block sizes, it would split the data into 3 blocks of 8 bytes, represented below with the hex values of each character:

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  ?  ?  ? |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 ?? ?? ??|
+-----------------------+-----------------------+-----------------------+
```

Those 3 characters at the end will be the padding that we need to add to the string. Let's see how this looks with each convention:

Each padded byte is the value of the total number of bytes needing to be added

This is the most popular method used throughout the industry, as it's easy during decryption to read these padding bytes and know, easily, what is a padding byte and how many you should be looking for.

In this case, each byte of padding is set to the value of the total number of bytes of padding. In this case, we have 3 bytes of padding, so we set the value to 03.

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  _  _  _ |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 03 03 03|
+-----------------------+-----------------------+-----------------------+
```

The first padded byte is 0x80, followed by zero bytes for the rest

For this one, we set the first byte of padding to 0x80 and all subsequent padding bytes to zero.

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  _  _  _ |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 80 00 00|
+-----------------------+-----------------------+-----------------------+
```

Each byte is set to zero except the last byte, which is the number of the total bytes of padding

This is a take on the first method, but we set all bytes of padding to zero, with the exception of the last byte, which should be equal to the total number of bytes of padding. In the case of our example, that will be 03.

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  _  _  _ |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 00 00 03|
+-----------------------+-----------------------+-----------------------+
```

All bytes are set to zero

All bytes of padding should be set to zero.

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  _  _  _ |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 00 00 00|
+-----------------------+-----------------------+-----------------------+
```

All bytes are set to spaces (0x20)

All bytes of padding should be set to spaces.

```
+-----------------------+-----------------------+-----------------------+
|T  h  i  s  _  i  s  _ |m  y  _  b  l  o  c  k |_  d  a  t  a  _  _  _ |
+-----------------------+-----------------------+-----------------------+
|54 68 69 73 20 69 73 20|6d 79 20 62 6c 6f 63 6b|20 64 61 74 61 20 20 20|
+-----------------------+-----------------------+-----------------------+
```

How padding works is something that we should understand when choosing a proper mode of operation, as it can be costly depending on the data that is to be encrypted. In each case of the padding, the decryption process needs to know how to properly determine which method was used, and how to properly disregard padding values.

In the case of the algorithms that we are going to use in the practical examples below, padding will be taken care of for us, so it's not something that we need to be concerned with in our implementations.

Block Cipher Modes of Operation

Understanding the different potential modes of operation is important when working with data encryption. A mode of operation is an algorithm that uses a block cipher (such as AES) to provide functionality such as confidentiality or data authentication for the block of information to be encrypted.

Within the case of our web operations, there are three main categories of modes that we will be working with:

- **Encryption**: This means, in general terms, data privacy / confidentiality. A potential attacker who has the generated ciphertext (the encrypted data) will not be able to get any information about the plaintext data, except for the length perhaps.
- **Authentication**: Authentication mode provides a mechanism for determining data authenticity. If a receiver obtains ciphertext or cleartext from a sender, they can determine if the data is genuine and was constructed by the sender.
- **Authenticated Encryption**: This includes both previous categories.

Now that we have an understanding of the different categories, let's look at some of the different NIST approved block cipher modes of operation. Many of these modes range heavily in how they function, but at the end of this section we list off the most popular industry standard modes.

If you want to protect the privacy of the data that you are sending, but don't necessarily need to authenticate the sending source of the data, then the mode under the encryption category will fill that need.

Table 7-1. Encryption Modes of Operation

Mode	Name	Description
ECB	Electronic Codebook	The simplest mode of operation. The data that is supplied for encryption will be divided into blocks. When encrypting / decrypting, these blocks will be processed individually. The main issue with this mode is that identical plaintext blocks are encrypted into identical ciphertext blocks, which creates a pattern that a potential hacker can exploit. One of the benefits on this mode is that the blocks can be processed in parallel, speeding up encryption / decryption.

Mode	Name	Description
CBC	Cipher Block Chaining	In this mode, each plaintext block is XORed against the previous ciphertext (encrypted) block, before then being encrypted itself. Since this is an iterative algorithm, an initialization vector needs to be supplied. In order to make each block unique, the initialization vector needs to supplied as the first block. One of the chief drawbacks is that blocks cannot be processed in parallel, since each block needs to be run sequentially. The other issue is that plaintext input needs to be a multiple of the block size (each block the same size), meaning that the initial data may have to be padded to bring it to that length.
OFB	Output Feedback	The output of OFB is what is called a stream cipher. The plaintext blocks are combined with random or pseudorandom characters to generate keystream blocks. These keystream blocks are then XORed with the plaintext blocks to get the end ciphertext. This mode also utilizes a supplied intialization vector. One of the benefits with this mode is that each block can be a different size, meaning that there is no need to pad the initial plaintext data. One of the drawbacks is that encrypting and decrypting blocks cannot be done in parallel, since encrypting each block depends on the previous blocks.
CFB	Cipher Feedback	CFB mode is very similar operationally to CBC. The main difference between these two is that CBC mode creates the ciphertext after the block cipher algorithm is run, while CFB generates it after we computer the XOR.
CTR	Counter	Counter mode, much like OFB, turns the block cipher into a stream cipher. The keystream that is used can be a function that produces successive values for the random data (a counter). The function producing the counter data just needs to ensure that the data does not repeat for a long period of time. The most common counter type is simply one that increments a number by 1 each time. Where CTR differs from OFB is that both CTR encryption and decryption can be run in parallel, since it doesn't generate the ciphertext by XORing the plaintext block against the previous ciphertext block (like in OFB).

If confidentiality is not required for the message being sent, that's where the authentication mode will come into play. For instance, say you just need to know that the message came from an approved sender, but the message itself is not sensitive in the way that you are concerned about data privacy, then these category of modes are probably best.

Table 7-2. Authentication Modes of Operation

Mode	Name	Description
CMAC	Cipher-based Message Authentication Code	CMAC mode is used for determining the authenticity and integrity of a message. It uses a block cipher algorithm in conjunction with a secret key to generate the resulting cipher. This mode is not heavily used.

In many cases you probably want to not only maintain the privacy of the data being transmitted, but also verify the source from which the data came, for additional security. In this case, the combined authenticated encryption modes are an excellent option.

Table 7-3. Authenticated Encryption Modes of Operation

Mode	Name	Description
CCM	Counter with CBC-MAC	This is the easiest combined mode of operation. As the full name suggests, CCM is a combination of the CTR and CBC modes of operation. This mode also includes the use of an initialization vector, and message authentication is done on the plaintext data. Encryption / Decryption cannot be run in parallel.
GCM	Galois/Counter Mode	GCM has been widely adopted due to its efficiency and performance. Like CCM, GCM uses an initialization vector, but the message authentication is done on the ciphertext as opposed to the plaintext data. Encryption / Decryption can also be run in parallel, unlike CCM.
KW / KWP / TKW	Key Wrapping	Permutations of proposed key wrap algorithm modes for encrypting and authenticating data. These modes are not widely used.

Even with a basic understanding of the above, it can be difficult to understand what the best modes of operations would be in each category. If we break them down into current industry standard use, there are the most popular modes of operation in each category:

- Encryption: CTR (for good parallelization / speed).
- Authentication: CMAC (because it's the only approved one, although it's not widely used in the industry).
- Authenticated Encryption: GCM (industry accepted standard).

Even though there are preferred modes by many in the industry, the mode that you choose to use should always come down to your needs on security, performance, scaling, and how each mode fits into those plans. Just because it's popular, doesn't mean it's right for you.

Now that we have an understanding of different modes of operation, let's apply these in practice with a few examples, starting with AES using the CTR encryption mode.

Using AES with CTR Encryption Mode

The sample code that we will go through below for using AES with CTR mode is available at *https://github.com/iddatasecuritybook/chapter7/blob/master/symmetric-crypto/aes-ctr.js*

We have all the our core basics to start looking at using a cipher algorithm (AES) with a particular mode of operation (CTR encryption in this case).

We're going to be using the standard Node Crypto package for this example, so there is nothing to pull down from npm. At the top of your node script, add the following initialization variable.

```
var crypto = require('crypto');
```

Now we need to set the variables that we are going to be working with in the example.

```
var text = "Encryption Testing AES";
var key = crypto.randomBytes(32);       //256 bit shared key
var iv = crypto.randomBytes(16);        //initialization vector - 16 bytes
var algorithm = 'aes-256-ctr';          //cipher and mode of operation
```

Going from top to bottom, these are:

- text: The data to be encrypted / decrypted.
- key: A 32 bytes shared key to be used by the crypto library for encryption / decryption. We use the `crypto.randomBytes(…)` method to generate that string.
- iv: The initialization vector, which should be a random 16 byte string. We use the `crypto.randomBytes(…)` method to generate that string.
- algorithm: The cipher algorithm and mode of operation to use. In this case we're using the AES cipher algorithm with the CTR mode of operation.

Next, we create the ciphertext by encrypting the data.

```
var cipher = crypto.createCipher(algorithm, key, iv);
var encrypted = cipher.update(text, 'utf8', 'hex');
encrypted += cipher.final('hex');
```

We first make a call to `crypto.createCipher(…)` to initialize the cipher that we want to use, passing in the algorithm / mode from above, the shared key, and the initialization vector.

Next, we use `cipher.update(…)` to update he cipher with data. We supply the data to be encoded, the input encoding (utf8), and the output encoding (hex).

cipher.update(data, input_encoding, output_encoding)

The input encoding (2nd parameter) should be one of *utf8*, *ascii*, or *binary*. If no input encoding is specified, the data (1st parameter) must be a Buffer. If a Buffer is specified as the data, input encoding will be ignored.

Lastly, we make the request to `cipher.final(…)`, passing in the output encoding type, to return the ciphertext.

Our plaintext input is now encoded. At this point there should be a few things known about the ciphertext and decryption:

- The ciphertext can now be transmitted to its end source (the receiver).
- The shared key and initialization vector must be known to the receiver to decrypt. The key should be a shared secret between the app and the host (sender and receiver), and the IV can be considered as a one-time use nonce, shared between both parties.

Now, let's say the cipher has been transmitted to the receiver, and they also have the key and initialization vector. We can now begin deciphering the ciphertext to extract our plaintext message.

```
var decipher = crypto.createDecipher(algorithm, key, iv);
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
```

We start by making a call to `crypto.createDecipher(…)`, passing along the same values that we did for creating the cipher during encryption.

We then call `decipher.update(…)`, passing in the ciphertext, the ciphertext encoding (in this case, hex), and the intended output encoding (in this case, utf8).

Lastly, we call `decipher.final(…)` with the output encoding type to retrieve our final decoded message.

Using AES with with GCM Authenticated Encryption Mode

The sample code that we will go through below for using AES with GCM mode is available at *https://github.com/iddatasecuritybook/chapter7/blob/master/symmetric-crypto/aes-gcm.js*

Now let's look at an example that is using the GCM joint authenticated encryption mode of operation. This one will be fairly similar as the above CTR mode example, but with an additional encryption / decryption step to handle the authentication piece that we didn't have in the CTR mode example.

We're again going to use the standard Node crypto package, so require that at the top of your Node script.

```
var crypto = require('crypto');
```

Next, we initialize the variables that we are going to be using.

```
var text = "Encryption Testing AES GCM mode";
var key = crypto.randomBytes(32);       //256 bit shared key
var iv = crypto.randomBytes(16);        //initialization vector - 16 bytes
var algorithm = 'aes-256-gcm';          //cipher and mode of operation
```

As before, these variables are:

- text: The data to be encrypted / decrypted.
- key: A 32 bytes shared key to be used by the crypto library for encryption / decryption. We use the `crypto.randomBytes(…)` method to generate that string.
- iv: The initialization vector, which should be a random 16 byte string. We use the `crypto.randomBytes(…)` method to generate that string.
- algorithm: The cipher algorithm and mode of operation to use. In this case we're using the AES cipher algorithm with the GCM authenticated encryption mode of operation.

Now, let's encode the data to get our ciphertext, and see the differences when we add the authentication piece.

```
var cipher = crypto.createCipher(algorithm, key, iv);
var encrypted = cipher.update(text, 'utf8', 'hex');
encrypted += cipher.final('hex');
var tag = cipher.getAuthTag();
```

We first make a call to `crypto.createCipher(…)` to initialize the cipher, passing in the algorithm / mode from above, the shared key, and the initialization vector.

Next, we use `cipher.update(…)` to update he cipher with data. We supply the data to be encoded, the input encoding (utf8), and the output encoding (hex).

We then create the ciphertext by calling `cipher.final(…)` with the output formatting.

This is where the authentication piece comes in, after we have the ciphertext, we now have to generate a tag by calling `cipher.getAuthTag(…)`. This will be a Buffer containing an authentication tag that has been computed from the cipher data. This will be used to authenticate the source of the ciphertext.

As of the writing of this text, getAuthTag only supports the GCM authenticated encryption mode.

As in the last example, the ciphertext should be transmitted to the receiver. The receiver should also have knowledge of the shared key, the initialization vector, and the authentication tag.

```
var decipher = crypto.createDecipher(algorithm, key, iv);
decipher.setAuthTag(tag);
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
```

We make our call to `crypto.createDecipher(…)`, passing along the same values that we did for creating the cipher during encryption.

Following that, we need to pass in the authentication tag to validate the source of the ciphertext. We do so by calling `decipher.setAuthTag(…)`, passing in the tag that was generated from the encryption step.

Next, we go back to the standard decipher techniques by calling `decipher.update(…)` to pass in the data, providing the ciphertext, ciphertext encoding (in this case, hex), and the intended output encoding (in this case, utf8).

Lastly, we call `decipher.final(…)` with the output encoding type to retrieve our final decoded message. The ciphertext will now be decoded and the source authenticated, allowing us to use the data at will.

Advantages, Disadvantages, and Uses of Synchronous Cryptography

As we close out our exploration of synchronous cryptography, it's a good idea to understand what it does well, and what it doesn't do well.

You have some advantages to using this method over its asymmetric counterpart:

- Security: When we are using a secure algorithm, such as the U.S. government designated Advanced Encryption Standard (AES) that we looked at above, the ciphertext that is produced is incredibly secure, and is considered essentially unbreakable with current computing standards.
- Speed: One of the main issues with asymmetric key cryptography is the complexity of the process that they need for encryption and decryption. With symmetric cryptography, using modes of operation that allow for parallel block processing for encryption and decryption, we have a fast processing mechanism.

If we flip the coin though, there are a few drawbacks to this methodology:

- Shared key: As you saw in the practical examples, we generate a single shared key that is used by the sender for encryption, as well as the receiver for decryption. You need to take great care in ensuring that this key can be shared between both parties, without being retrieved by an attacker. Should this key be obtained by an illicit third party, they now have access to all encrypted data that is using that shared key. This means that the amount of damage that may be caused by using this method is typically quite high.

With all said and done, one of the best methods for using this type of cryptography is when you're encrypting and decrypting your own data, when you have safe, secure access to shared keys between endpoints. All in all, if you have a safe way to share the key between yourself and an unknown party, you probably don't need to be using your own encryption in the first place.

APPENDIX A

GitHub Repositories

Jonathan LeBlanc and Tim Messerschmidt

We are well aware that mistakes might have slipped into the code that were not spotted throughout the review period, that code needs to be updated in order to reflect current trends or that those libraries and modules we build upon are deprecated in favor of something better. This is why we have created an organization on GitHub[1] that holds all code we present throughout this book. You will find the complete OAuth 2 and OpenID Connect server, the client that interacts with those service providers and all those small snippets we use in order to demonstrate certain features.

In case you have any questions or want to contribute - please feel free to create an issue on GitHub or fork the affected project, make the change and send us a pull request. We promise to proactively improve the code over time in order to ensure longevity of the information we conveyed here.

1 *http://github.com/iddatasecuritybook*

APPENDIX B

Technical Preconditions and Requirements

In the following Appendix we'd like to take the time to explain some basic concepts around Node and add extra material that supplement's the book's content.

On ES6 / ES2015

You'll notice that we are using ES5 syntax as a base for the code throughout this book. The simple reason is that we wanted to use the more common syntax in order to ensure that everyone is able to comfortably read through this book.

We highly encourage you to try out ES6 by using compilers / transpilers such as Babel[1]. Node is slowly adopting ES6 - in the meantime you can prepare your application and ensure you are ready to deploy to production for when ES6 is fully supported.

Setting up your Node.js Environment

This book assumes a working Node.js environment and the ability to install various modules using **npm**. In this section we'd like to run you through setting up Node in order to ensure that you will be able to follow the book.

First of all please install Node by either downloading and running the installer from nodejs.org's downloads section[2] or leveraging a package manager such as brew[3], pac-

1 *http://babeljs.io/docs/usage/cli/*

2 *http://nodejs.org/en/download/*

3 *http://blog.teamtreehouse.com/install-node-js-npm-mac*

man or apt-get[4]. Verify that the installation worked by typing `which node` or `node -v` into your terminal - this should display either the path to your node executable or your local node version.

> The download from nodejs.org always serves the most recent stable version of Node and should be favored over alternative ways of installing Node whenever possible.

Once Node is set up, we can proceed with installing Express by utilizing `npm`. To see if your environment is working correctly, simply enter `npm` in your terminal - you should see a brief explanation about using the command.

Managing Node versions or alternative installations

By using the installer obtained from nodejs.org, the newest stable version of Node.js will be installed on your machine. Sometimes you might have to use a certain other version of Node or even switch between multiple versions based on the project you are working on. `nvm` (which stands for Node Version Manager) is a community project that allows for doing exactly this.

You can use the install script for cURL by running the following command:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh | bash
```

Assuming you'd like to install version 5.7.0 of Node.js (the latest stable version of Node), you'd run the following command:

```
nvm install 5.7.0
```

Once the installation is done, run `nvm use 5.7.0` and your environment is setup: `5.7.0` will be your system's default Node.js version.

Should you be required to run multiple versions of Node, you have the choice of either running the specific version by using the command-line interface `nvm run 5.7.0` or by setting up a project-specific `.nvmrc`-file containing the target version number.

Installing the Express generator

Assuming the set-up works as intended, we can proceed by installing and using the Express generator. This generator allows for scaffolding our project and creating a reasonable structure.

4 *http://nodejs.org/en/download/package-manager/*

```
npm install -g express-generator
```

You will notice that we use the `-g` option. This implies that we want to install the generator globally and not just as a module for our current project.

To verify that the Express generator was set up correctly, we type the following command to see the generator's usage information:

```
express --help
```

More documentation around the generator and Express can be found on ExpressJS.com[5].

Setting up Express

In case you'd like to avoid using a generator to set up your first Express project, you can start by simply following these instructions:

1. Create a new folder with `mkdir projectname`
2. Navigate towards the folder by running `cd projectname`
3. Create a `package.json` file by running `npm init` in your project's root folder. Don't worry about fleshing out the content yet - the next section will go into `package.json` in more detail.
4. Run `touch app.js` (or whatever name you've defined in step 4) to create your application's entry point. Here we'll define many of the modules our project ends up using, create the server itself and initialize the middleware our project might rely on (see Chapter 4 authorization middleware).
5. Finally run `npm install express --save` to install the most important module for our project: Express itself.

After you've taken care of these five steps we're ready to implement a very basic Express application that you can then build out in order to achieve additional functionality:

```
var app = require('express')();

app.get('/', function(req, res) {
  res.send('Hello from Express!');
});

app.listen(3000, function() {
  console.log('App active on port 3000');
});
```

5 *http://expressjs.com/en/starter/generator.html*

The above example presents the beauty of Express - with mere 9 lines of code we can run a simple web-server. By navigating to `localhost:3000` you'll be presented with a hearty `Hello from Express!`.

Run the application with the following command: `node app.js`. Congratulations!

You can extend on this by starting to serve static resources (such as your site's CSS and JavaScript or images) by modifying the above example:

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/', function(req, res) {
  res.send('Hello from Express!');
});

app.listen(3000, function() {
  console.log('App active on port 3000');
});
```

You can also specify that static resources are served through another folder structure such as `/resources` by using the following method:

```
app.use('/resources', express.static('public'));
```

It is safer to declare the folder by using the absolute path to the folder: `__dirname + '/public'`. `express.static` uses relative paths - this leads to issues when running the Node process from another folder:

```
app.use('/static', express.static(__dirname + '/public'));
```

Creating and maintaining your package.json file

In case you've been using the Express generator you will notice that a file called `package.json` got generated for you. Otherwise go ahead and create one by running `npm init` in your terminal.

`package.json` contains information about module dependencies, the project's authors, versioning, and a section called `scripts` containing commands for `npm`. In the below example case running `npm start` or `npm run start` starts a new instance of our newly generated node server.

```
{
  "name": "book",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
    },
    "dependencies": {
      "body-parser": "~1.13.2",
      "cookie-parser": "~1.3.5",
      "debug": "~2.2.0",
      "express": "~4.13.1",
      "jade": "~1.11.0",
      "morgan": "~1.6.1",
      "serve-favicon": "~2.3.0"
    }
  }
```

You can also extend the scripts section for other tasks like running eslint:

```
  "scripts": {
    "start": "node ./bin/www",
    "lint": "eslint app.js lib/** routes/**"
  },
  ...
```

Damon Bauer wrote a great blog post on using npm scripts for a variety of tasks that normally would be covered through task runners like grunt, gulp or broccoli[6]. It's a highly recommended read in case you are looking for further inspiration.

By running `npm install` all entries in the `dependencies` section will be retrieved. `npm update --save` ensures that your project uses the latest version of each dependency in the `dependencies` section and updates `package.json` accordingly. The site npmjs.com allows to search for modules or browse popular choices[7] - you will notice that `npm` itself is a module that can be updated using `npm` itself.

Additional sections - such as `bugs` - can be defined to help developers understand how to interact best with the project, how to contact the developers in case they want to get in touch, or under which license the project got published. Check out the browsenpm.org for an interactive overview about all possible `package.json` sections[8].

Application Configuration

Environment variables are a sensible way of handling configuration details - such as database passwords or third party API credentials - without hardcoding them in your application's code. The Twelve-Factor App methodology[9] defines this as separation of code and configuration and sees the benefit of being able to quickly change between different deployment targets such as production environments.

6 *http://css-tricks.com/why-npm-scripts/*

7 *http://www.npmjs.com*

8 *http://browsenpm.org/package.json*

9 *http://12factor.net/config*

The module dotenv[10] was designed to specifically cater to this use case and utilizes a configuration file called `.env` (located in your project's root folder) that stores information in the `NAME=VALUE` format:

```
MONGO_DB=mongodb://localhost/database
MONGO_USER=tim
MONGO_PW=sloths-are-more-awesome-than-monkeys
```

You can access these details by simply loading the module and calling the `config` method. You will notice that `process.env` is going to be populated with the information from your `.env` file.

```
require('dotenv').config();

var mongoose = require('mongoose');
mongoose.connect(process.env.MONGO_DB);
```

dotenv allows to pass different configuration options - such as `path` (in case you require a different location for your configuration file), `silent` (which suppresses warnings when no `.env` file can be found), or `encoding` (the default is `utf8`).

Version Control and Configuration Files

Make sure to exclude your dotenv configuration files by adding them to your project's `.gitignore` file. Please also check out dotenv's FAQ for more information[11] on this matter.

Working with JSON / URL Encoded Bodies in Express

As of Express 4.0, working with JSON and URL encoded bodies has changed slightly, and requires an additional setup step to be able to work with that data when it is POSTed to your server. The body-parser npm module, when used in the following format, will allow your server to support those entities:

```
var bodyParser = require('body-parser')
var app = require('express')();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));
```

10 *http://github.com/motdotla/dotenv*

11 *http://github.com/motdotla/dotenv#should-i-commit-my-env-file*

Glossary

2FA / Two-factor authentication
: The process of using a secondary means of identification during a login or user discovery step. Typically, your first authentication system is a username and password, then your second factor of authentication may be a code provided by SMS to a registered phone number, a registered fingerprint, code via email, etc.

Ciphertext
: Unreadable output of an encryption algorithm

Cleartext
: Human readable data that is transmitted or stored unencrypted

EFF
: The Electronic Frontier Foundation. A nonprofit dedicated towards protecting user privacy and civil liberties in the digital world.

Entropy
: In the context of identity, this concerns the amount of information that is discoverable about a user, determining the likelihood that the user is who they say they are.

MFA
: Multi-factor authentication is a means of user identification that requires more than one method of authenication (username & password, SMS code, email code, fingerprint, etc).

NIST
: The National Institute of Standards and Technology, a unit of the U.S. Commerce Department. NIST promotes and maintains measurement standards and maintains active programs for encouraging and assisting industry and science to develop and use these standards.

OWASP
: The Open Web Application Security Project is a community that maintains tools, documentation, and guides in the field of web application security, and is considered a forefront standard in the space.

Plaintext
: Human readable data that is supplied to the encryption algorithm

Security-layering
: The practice of using multiple security mechanisms in a stacked approach to protect identity, data, and resources.

U2F
: Universal 2nd Factor is an open authentication security standard that aims to strethen/ simplify two-factor authentication using specialized USB or NFC devices based on similar security technology found in smart cards.

UAF

The Universal Authentication Framework protocol defines the process for passwordless user experiences. This may be through voice commands, facial recognition, or some other like standard.

Index

About the Author(s)

John Doe does some interesting stuff...

Colophon

The animal on the cover of *FILL IN TITLE* is *FILL IN DESCRIPTION*.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *FILL IN CREDITS*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.