Софтуерната сигурност е защита срещу: загуба на данни, прекъсване на услуга, изтичане и/или несъответствие на данните.

Загуба на данни – основните данни се губят поради нарушение на сигурността, т.е. нападателят получава достъп до основната база данни и изтрива записите. Пример: [Hacker deleted all data from VFEmail Servers, including backupsSecurity Affairs](https://securityaffairs.co/81030/hacking/vfemail-destructive-cyberattack.html)

Прекъсване на услугата - Дейността на системата е нарушена поради действия на атакуващия, т.е. нападателите организират атака, водеща до отказ на услугите на системата. Пример: 

Изтичане на данни - Основните данни се крадат и се предоставят на неоторизирани получатели, т.е. нападателите получават достъп до базата данни и крадат важна информация, като например кредитни карти. Пример: [Marriott Hackers Stole Data On 500 Million Guests -- Passports And Credit Card Info Included (forbes.com)](https://www.forbes.com/sites/thomasbrewster/2018/11/30/marriott-admits-hackers-stole-data-on-500-million-guests/?sh=50f10ba46492#786737086492) 

Несъответствие на данните - данните се манипулират от неоторизирани нападатели и стават непоследователни, т.е. нападателите се представят за някой друг и извършват неоторизирани действия.

Сигурността на софтуера включва терминология, използваща се в много дискусии. Таблица х.х. описва някои основи термина.

|  |  |  |
| --- | --- | --- |
| Име | Описание | Пример |
| Заплаха (Threat) | Събитие, което, ако се случи, ще доведе до инцидент със сигурността. | SQL Injection, DDOS атака |
| Атака (Attack) | Действително изпълнение на заплаха от нападател(и). | Злонамерен потребител въвежда SQL Injection или група нападатели организират DDOS атака. |
| Уязвимост (Vulnerability) | Проблем в системата, който може да бъде използван от атакуващия, например за компрометиране на данните. | Неправилно конфигурираната защитна стена излагаща вътрешните системи. |
| Удостоверяване (Authentication) | Установяване на самоличността на потребител (човек или ядошкеи) въз основа на надеждни  Механизми. | потребителско име / парола, СМС или биометрична идентификация |
| Упълномощаване (Authorization) | Установяване какво е разрешено да прави даден потребител в системата. | Потребителят X има право да създаде нова поръчка. Потребителят X НЯМА право да отмени съществуваща поръчка. |

Процес на защитена архитектура гарантира, че системата е възможно най-добре защитена. Той включва всички фази на системата, ръководен от мениджъра на проекта и архитекта на системата. Процесът се състои от 5 етапа:

Моделиране на заплахи - Процес за идентифициране на потенциални заплахи за система. Включва почти всички от софтуерния екип, като може да използва официални методи и инструменти. Процесът по моделиране включва три основни въпроса: 1. Какво се изгражда? 2. Какво може да се обърка? 3. Как може да се предотврати това?

Сигурна архитектура – интегрира защитата на сигурността в ядрото на архитектура. Основава се на концепцията за периметър на сигурност. Отнася се за граница, която разделя вътрешната мрежа, системите и данните на организацията от външните мрежи като интернет. Традиционните мерки за сигурност като защитни стени, системи за откриване на проникване и антивирусен софтуер често се разполагат на ръба на тази граница, за да защитят вътрешната среда от външни заплахи. Въпреки това, еволюция в технологиите създава динамичен цифров „периметърът“, който не е фиксирана линия, а по-скоро постоянно променяща се граница.

Тази промяна доведе до нови парадигми, включително:   
Zero Trust Network Architecture и Secure Access Service Edge (SASE)

Мрежова архитектура с нулево доверие: Този модел работи на предпоставката, че заплахите могат да идват както отвън, така и отвътре в периметъра на сигурността, следователно „не се доверявайте на никого“ се превръща във водещ принцип. Zero Trust налага стриктна проверка на самоличността на всеки човек и устройство, които се опитват да получат достъп до ресурси в частна мрежа, независимо дали се намират в мрежата или извън нея.

Secure Access Service Edge (SASE): Този модел съчетава мрежова сигурност и възможности за широкообхватна мрежа (WAN) в една облачна услуга, за да позволи сигурна и бърза облачна трансформация. Той опростява управлението и внедряването на мрежова сигурност, като предоставя и двете като унифицирана услуга.

Жизненият цикъл на защитена разработка (SDLC) - подход за изграждане на приложения и софтуерни системи, който включва практики и принципи за сигурност от самото начало и през всички фази на процеса на разработка. SDLC може да се разглежда като интеграция на традиционни процеси на разработка с практики за сигурност, за да се гарантира, че уязвимостите в сигурността се идентифицират и адресират възможно най-рано. Общата цел на SDLC е да намали риска от уязвимости на сигурността в крайния продукт чрез интегриране на сигурността в процеса на разработка, вместо да се опитва да я добави по-късно като последваща мисъл. Това е по-проактивен подход към сигурността и има тенденция да води до по-сигурни продукти, тъй като третира сигурността като основен компонент на разработката на системата и софтуера.

Ориентирано към сигурността тестване - включва различни видове тестване, насочени към потенциални уязвимости. Някои от тях са статичен и динамичен анализ. Статичния анализ включва анализиране на код без неговото изпълнение, като се използват автоматизирани инструменти за намиране на потенциални уязвимости, свързани с несигурни практики за кодиране. Динамичен анализ е тип тестване включва анализиране на приложение, докато работи. Това може да включва тестване на откритите интерфейси на приложението за уязвимости (fuzz testing) или тестване на системата за уязвимости, които се появяват по време на работа.   
Друг вид тестване е реален тест за сигурността на системата, при който етични хакери се опитват да използват уязвимостите във вашата система по контролиран начин. След извършване на тестването, трябва да се анализират резултатите, като същевременно се идентифицират всички уязвимости, открити по време на процеса на тестване. След това трябва да се приоритизират тези уязвимости въз основа на фактори като потенциално въздействие, лекотата, с която могат да бъдат използвани, и критичността на системния компонент, където са открити. Стратегии за справяне с тези уязвимости, може да включват пренаписване на код, промяна на системните конфигурации или внедряване на нови контроли за сигурност.

След анализиране резултатите от тестовете, трябва да се сравнят с модела на заплаха, включвайки: проверка дали откритите по време на тестването уязвимости са включени във модела, проверка дали най-сериозните уязвимости съответстват на най-вероятните заплахи, дали определени типове заплахи от модела не са идентифицирани при тестване. Целта е не само да се открият и коригират уязвимостите, но и да се подобри цялостното разбиране за сигурността на системата.

Еволюцията на разработването на приложения и необходимостта от сигурни методи за удостоверяване, които да съответстват на променящите се технологии. В резюме, може да отбележим няколко етапа:

В миналото приложенията бяха предимно дебели клиентски приложения (като Windows Forms или WPF приложения) или сървърни уеб приложения, като и двете не бяха базирани на услуги. Тези приложения често използваха идентификационни данни на Windows или удостоверяване на формуляри и тяхната сигурност беше сравнително ясна поради изолацията им в рамките на един домейн или интранет на компанията.

Пейзажът започна да се променя с възхода на ориентирани към услуги приложения, често направени с помощта на рамки като Windows Communication Foundation (WCF). Въпреки това, това все още са обикновено дебели клиентски приложения в рамките на фирмен домейн или комуникации между сървъри, като WCF се справя с повечето проблеми със сигурността.

Тъй като ландшафтът за разработка на приложения продължи да се развива, ние започнахме да изграждаме уеб API и RESTful API, които не винаги се намират в същия домейн или под наш контрол. Приложенията започнаха да се интегрират с различни API, включително услуги на трети страни като Google Maps.

С преминаването към уеб приложения от страна на клиента, мобилни приложения и API, комуникиращи с други API, традиционните мерки за сигурност като поддържане на API в рамките на стените на компанията или използване на удостоверяване на формуляри станаха недостатъчни. Не можем да защитим нашите API, като просто ги запазим вътрешни, особено за мобилни или клиентски приложения, които трябва да взаимодействат с публичен уеб API.

В отговор разработчиците започнаха да използват система за сигурност, базирана на токени. Това включва изпращане на токени вместо комбинации от потребителско име и парола с всяка заявка за API. Токените представляват потребителско съгласие и могат да бъдат предадени на API за разрешаване на достъп. Създаването и безопасното доставяне на тези токени обаче повдигна нови предизвикателства.

Първоначално решение беше да се изградят услуги за токени, приемащи потребителско име и парола и връщане на JSON Web Token (JWT). Този метод е подобрение, но все още изисква споделяне на поверителна информация с приложението или API и включва много персонализирано внедряване.

Пасажът завършва с идентифициране на необходимостта от централен доставчик на идентичност, който може да се справи с общи задачи, свързани с идентичността, и доказан протокол, който се справя с общи проблеми, които може да пренебрегнем, когато сами създаваме услуги за токени.

Преди това приложенията отговаряха за собственото си удостоверяване на потребителя, включително екрани за влизане, управление на потребителите, правила за пароли и т.н. Днес този подход е по-малко предпочитан, защото децентрализира и усложнява сигурността.

Ролята за удостоверяване на потребител сега обикновено се управлява от централизиран доставчик на идентичност (IdP). Тези IdP често се справят и със задачи за управление на акаунти. Чрез централизирането на тези задачи системата става по-управляема, сигурна и многократно използваема в различни приложения.

Има няколко причини да предпочетете централизирани системи за управление на идентичността и достъпа. Те намаляват вероятността от грешки, по-лесни са за актуализиране и тестване и могат да управляват набор от задачи от регистрация на потребител до заключване на акаунт.

Пасажът дава пример за сигурност с парола, за да илюстрира необходимостта от централизация. Безопасното съхраняване на пароли еволюира от основно криптиране до методи за разтягане на ключове като Argon2 или bcrypt. Най-добрата практика е тази функция да се централизира, така че когато стандартите се развиват, те да могат да се актуализират на едно място.

В допълнение към обработката на пароли, IdPs могат да управляват други средства за удостоверяване, като сертификати или удостоверяване от втори и трети фактор. По-ефективно е тези опции да се управляват централно, отколкото да се налага да актуализирате всяко клиентско приложение поотделно.

Преминаването към централизация предоставя решения на първия проблем: преместване на процеса на удостоверяване на потребителя от отделни приложения към ниво IdP.

След като е налице централизиран IdP, на това ниво може да се добави код за обработка на крайни точки за влизане или токени. Това обаче не решава напълно проблемите, свързани с местните токен услуги, тъй като все още може да доведе до преоткриване на колелото.

OAuth 2.0 и OpenID Connect са протоколи, използвани за удостоверяване и оторизация в интернет, позволяващи на потребителите да демонстрират своята самоличност и привилегии за достъп.

OAuth 2.0 е протокол за оторизация, който позволява на приложенията да получат ограничен достъп до потребителски акаунти в HTTP услуга, като Facebook, GitHub и DigitalOcean. Той работи, като делегира удостоверяване на потребителя на услугата, която хоства потребителския акаунт и упълномощава приложения на трети страни за достъп до потребителския акаунт.

OAuth 2.0 се фокусира единствено върху оторизацията, осигурявайки защитен достъп до ресурси, но не и директно удостоверяване на потребителите. Това означава, че протоколът не проверява кой е потребителят, но проверява дали потребителят има разрешение за достъп до ресурсите или не.

OAuth 2.0 може да се използва за множество случаи на употреба, като например:

„Код за оторизация“ за приложения, работещи на уеб сървър.

„Парола“ за влизане с потребителско име и парола (не се препоръчва).

„Клиентски идентификационни данни“ за достъп до приложението.

"Implicit" преди се препоръчваше за клиенти без тайна, но беше заменен от PKCE (Proof Key for Code Exchange).

Основната цел на OAuth 2.0 е да упълномощи клиентско приложение за достъп до API чрез токени за достъп, които клиентът може да поиска и след това да използва, за да получи достъп до API. OAuth 2.0 взема предвид различните възможности на различни типове приложения (напр. уеб приложения от страна на сървъра спрямо приложения от страна на клиента) и предоставя насоки как всяко може да получи сигурно токени.

Чрез внедряването на OAuth 2.0 едно приложение може да преодолее няколко общи предизвикателства пред сигурността, като изтичане на токена и доставка на токени до различни клиентски приложения. Той замества необходимостта от персонализирани решения за сигурност със стандартизирани крайни точки и насоки за използване.

Приложенията, съвместими с OAuth 2.0, често се наричат доставчици на идентичност, сървъри за оторизация или услуги за маркери за сигурност. Добре известни примери включват Identity Server, Azure AD, Ping, Okta, Auth0, WSO2 Identity Server и TrustBuilder.

OAuth 2.0 обаче решава само половината от проблема. Той е проектиран да разрешава достъп до API, а не да удостоверява потребителите за влизане в приложението. Тук се намесва OpenID Connect, протокол, изграден върху OAuth 2.0. OpenID Connect допълва пъзела, като предоставя удостоверяване на потребителя върху оторизацията на OAuth 2.0.

OpenID Connect, често наричан само OIDC, е прост слой за идентичност върху протокола OAuth 2.0. Той позволява на клиентите да проверят самоличността на крайния потребител въз основа на удостоверяването, извършено от сървър за оторизация.

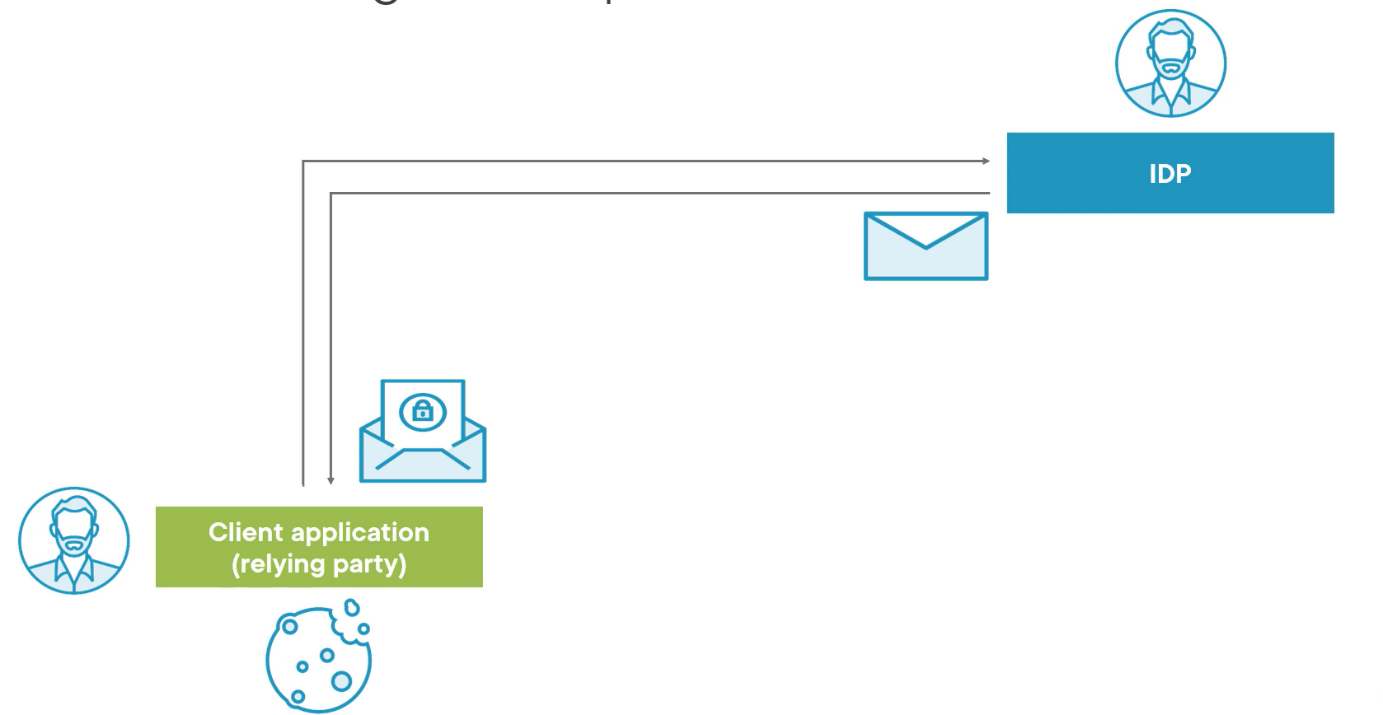
Казано по-просто, OpenID Connect е свързан с удостоверяването на потребителя: помага да се отговори на въпроса „Кой е този потребител?“. Той добавя ID токен към OAuth процеса, който носи информация за удостоверения потребител. Този ID токен е JSON Web Token (JWT), който съдържа информация (искания) за удостоверения потребител.

OIDC въвежда нов тип токен, ID токен, който е JWT и предоставя начин за директно удостоверяване на потребител. Това контрастира с OAuth, който предоставя само разрешение с токен за достъп.

OpenID Connect може да се използва за много случаи на употреба, включително обединяване от други доставчици, корпоративно SSO, мобилни приложения и javascript приложения от страна на клиента. Той предлага редица стандартни функции, включително откриване на доставчик на OpenID, динамична регистрация на клиент, режим на публикуване на формуляр, управление на сесии и други.

В обобщение, OAuth 2.0 се използва за оторизация, за да позволи на потребителите да делегират разрешения за своите данни, докато OpenID Connect се използва за удостоверяване, за да се потвърди кой е потребителят. Те често работят заедно, където OpenID Connect разширява OAuth 2.0, за да осигурят цялостно решение за удостоверяване и оторизация.

OpenID Connect работи: Разчита на токени за идентичност, за да потвърди самоличността на потребителя. Клиентското приложение или „доверяващата се страна“ изпраща заявка до доставчика на самоличност, който удостоверява идентификационните данни на потребителя и връща токен за самоличност. Този токен, съдържащ удостоверимата самоличност на потребителя, се валидира от клиентското приложение и след това се използва за създаване на ClaimsIdentity, обикновено съхраняван в криптирана бисквитка.



Типове клиенти: OpenID Connect включва различни приложения като уеб приложения от страна на сървъра, уеб приложения от страна на клиента и мобилни приложения. Той дефинира два типа клиенти - поверителни клиенти (способни да поддържат сигурно идентификационни данни като уеб приложения, работещи на сървър) и публични клиенти (неспособни да поддържат сигурно идентификационни данни като JavaScript или мобилни приложения).

Потоци и крайни точки на OpenID Connect: OpenID Connect използва различни потоци (последователности от заявки и отговори) и крайни точки за сигурно получаване на токени за самоличност. Ключовите крайни точки включват крайна точка за оторизация (за получаване на удостоверяване и/или оторизация), крайна точка за пренасочване (за връщане на код за оторизация или токени към клиентското приложение) и крайна точка за токени (за програмно изискване на токени).

Поток от най-добри практики: Понастоящем потокът от най-добри практики както за поверителни, така и за публични клиенти е потокът на кода за оторизация с PKCE (Ключ за доказване за обмен на код) защита. Този поток връща код за оторизация като краткотрайни идентификационни данни за еднократна употреба и токените се връщат от крайната точка на токена. За поверителни клиенти заявките към крайната точка на токена трябва да бъдат удостоверени, позволявайки дълготраен достъп чрез токени за опресняване. Публичните клиенти, които не могат да съхраняват сигурно идентификационни данни, имат ограничен дълготраен достъп.

Значение на правилния поток: Изборът и прилагането на правилния поток е от решаващо значение за предотвратяване на потенциални уязвимости в сигурността. Прилагането на грешен поток може да изложи значителни рискове за сигурността.

Променящ се характер на най-добрите практики: Документът подчертава, че най-добрите практики в сигурността се развиват бързо. Например клиентските приложения, базирани на потребителски агент, като Angular, React и т.н., се отдалечават от обработката на потока на OpenID Connect в браузъра, като сега предпочитат да го обработват на ниво сървър.

Потокът от кодове за оторизация е жизнеспособна опция за нашето приложение, стига да е защитено с PKCE. Стъпките в процеса са:

Първоначално искане за удостоверяване се прави до крайната точка за оторизация на доставчик на идентичност (IDP). Тази заявка включва параметри като client\_id, URI за пренасочване, заявени обхвати и response\_type.

Respon\_type в този случай е зададен на „code“, което показва използването на потока на кода за оторизация и че кодът за оторизация ще бъде върнат чрез пренасочване на браузъра.

След като потребителят предостави своите идентификационни данни (напр. потребителско име и парола), IDP знае кой е потребителят, но клиентското приложение не знае. Потребителят може също да бъде помолен за съгласие за споделяне на конкретна информация с приложението.

След удостоверяване, IDP връща код за оторизация чрез URI пренасочване (комуникация на преден канал). Кодът е краткотраен токен, свързан с потребителя, който е влязъл и действа като доказателство за удостоверяване.

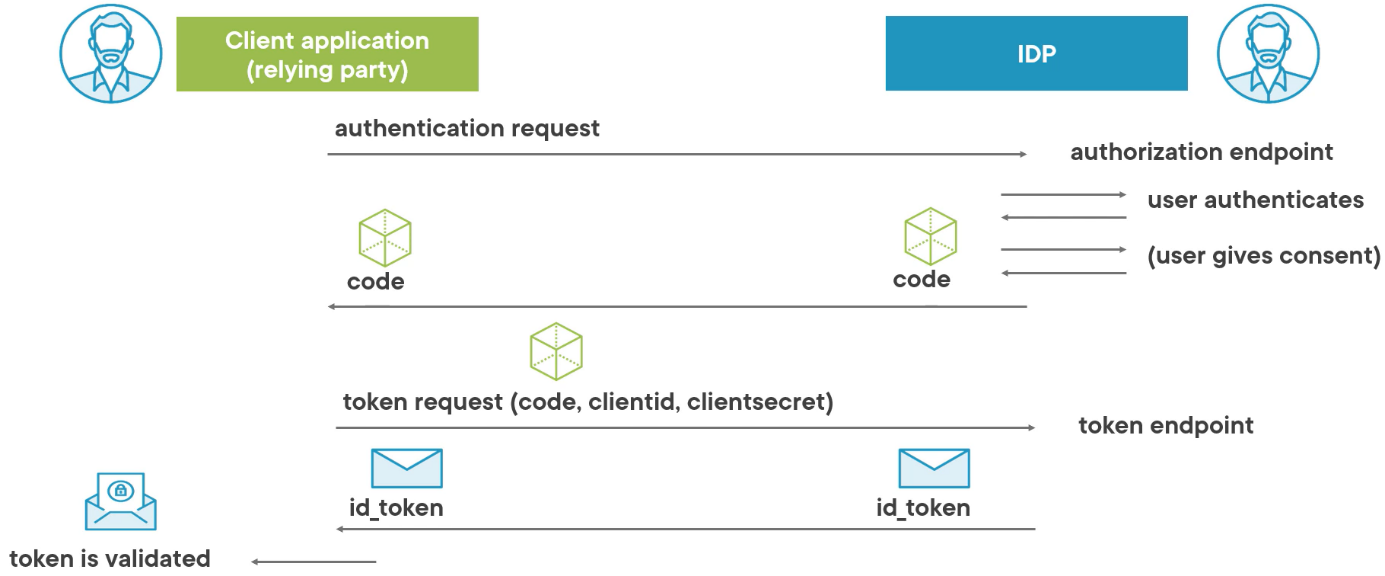
След това клиентското приложение използва този код, за да направи HTTP заявка от сървър към сървър към крайната точка на маркера на IDP (комуникация по обратен канал). Тази стъпка също включва удостоверяване на клиента и е по-сигурна, тъй като не се вижда от браузъра.

Ако всичко е наред, IDP генерира токен за самоличност и го изпраща на клиента. След валидиране на токена, клиентът може да извлече идентификатора на потребителя от него.

Валидираният токен служи като доказателство за самоличността на потребителя и след това клиентското приложение може да създаде самоличност за заявки въз основа на информацията за токена и да влезе в приложението.

Документът също така подчертава значението на използването на обратния канал за доставка на токени, за да се гарантира по-голяма сигурност, минимизиране на рисковете от изтичане на токени и следване на принципите на защита в дълбочина в сигурността.

Накрая се отбелязва, че въпреки че всички тези стъпки могат да бъдат извършени ръчно, има междинен софтуер, който опростява този процес, който ще бъде разгледан в следващата демонстрация



Потокът на кода за оторизация, макар и ефективен, е податлив на атаки с инжектиране на код, при които нападателят може да отвлече сесията на потребителя, като придобие кода за оторизация на потребителя. Метод за смекчаване на този риск за сигурността е чрез прилагането на стандарта Proof Key for Code Exchange (PKCE).

PKCE работи, като въвежда допълнителен слой на сигурност за всяка заявка към крайната точка за оторизация. Клиентското приложение генерира тайна за всяка заявка. След това тази тайна се потвърждава при извикване на крайната точка на токена и токените се връщат само когато тайната съвпада, което затруднява нападателя да отвлече сесията без достъп до тайната.

За да включите PKCE в потока на кода за оторизация:

Клиентското приложение генерира произволен низ, наречен code\_verifier.

Този code\_verifier се хешира, за да се получи code\_challenge.

Създава се заявка за удостоверяване, която включва този code\_challenge и се изпраща на доставчика на самоличност (IDP).

IDP съхранява code\_challenge и продължава с удостоверяването на потребителя.

След успешно удостоверяване, IDP пренасочва обратно към уеб приложението с кода за оторизация в URI.

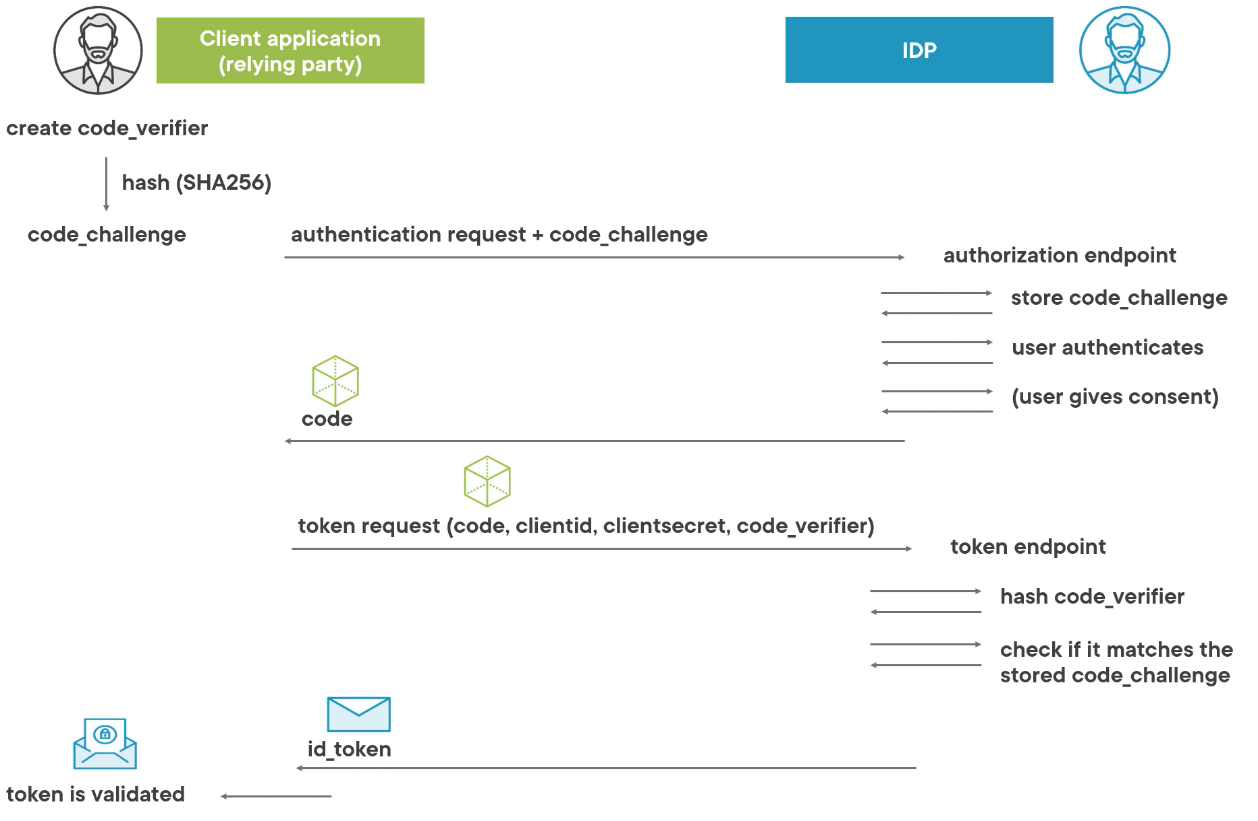
След това клиентското приложение извиква крайната точка на токена, удостоверява с помощта на clientid и secret и предава кода за оторизация и code\_verifier.

IDP хешира получения code\_verifier и проверява дали хешът съвпада със съхранения code\_challenge.

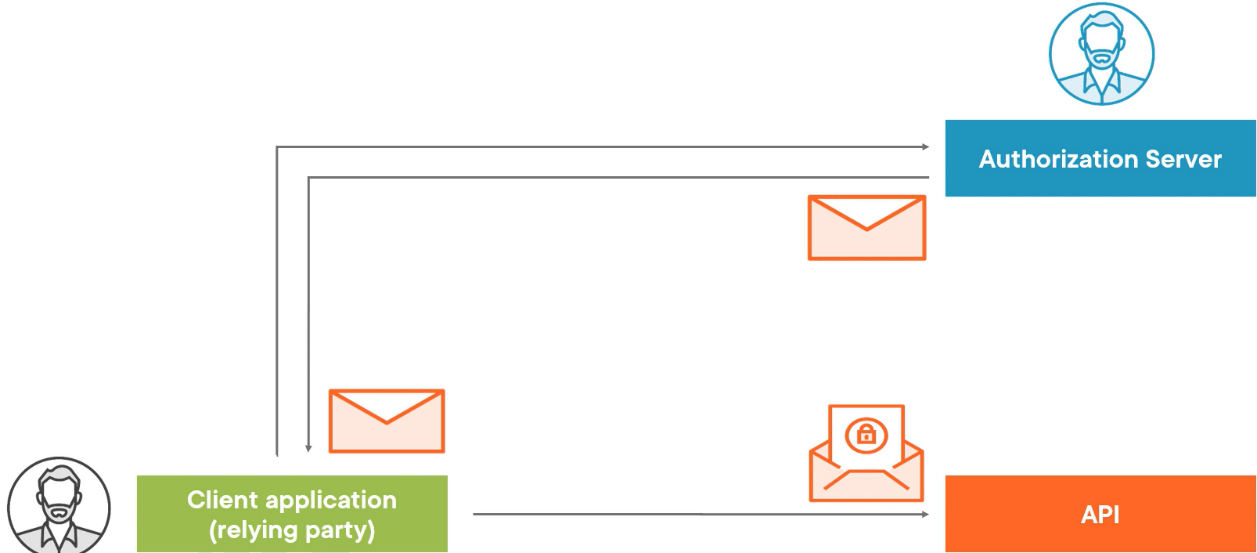
Ако съвпада, IDP потенциално връща токени (приемайки токени за идентичност в този случай).

Клиентското приложение валидира токена. Ако проверката е успешна, от нея се създава самоличност на искове и потребителят влиза в приложението.

Този подход осигурява ефективна защита срещу атаки с инжектиране на код, което прави потока на кода за оторизация с PKCE защита по-сигурен.



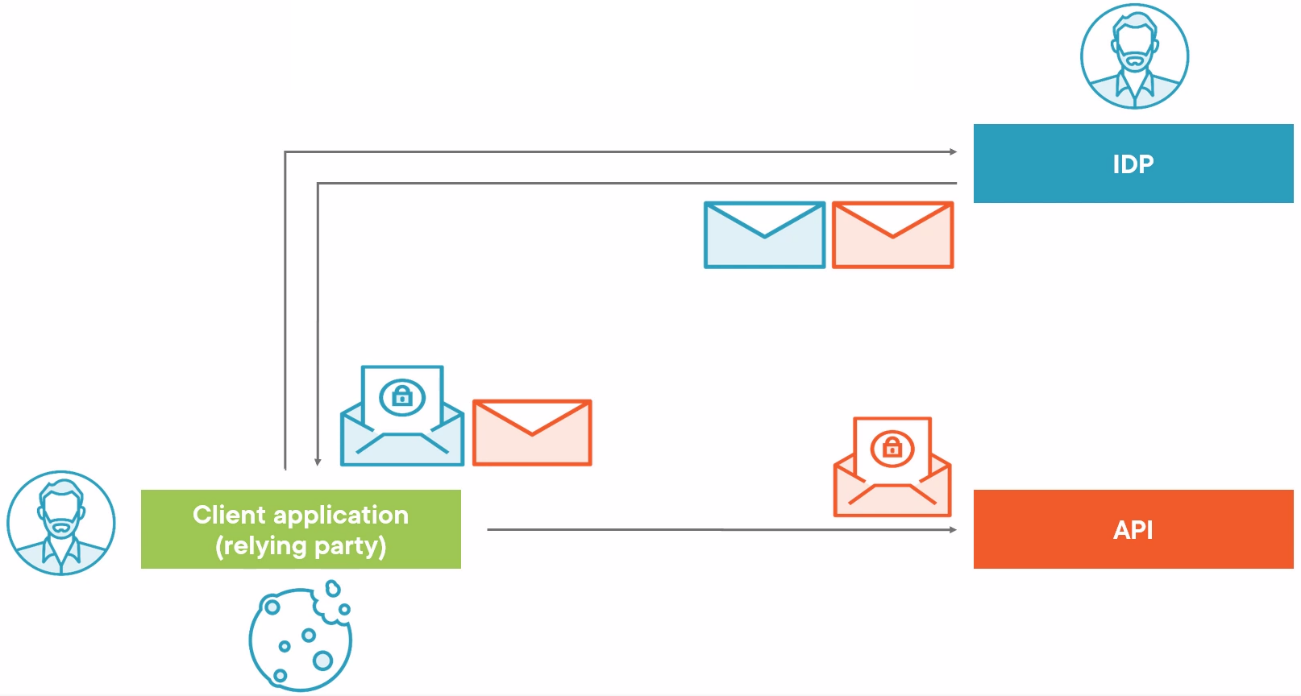
OAuth 2 е предназначен за делегирано оторизиране, което включва оторизиране на достъп до ресурси като API. Когато клиентско приложение се нуждае от достъп, то пренасочва потребителя към сървъра за оторизация. Потребителят се удостоверява, обикновено чрез потребителско име и парола, като дава възможност на сървъра за оторизация да ги идентифицира. След това сървърът генерира и подписва маркер за достъп, представляващ съгласието на потребителя клиентското приложение да има достъп до техните данни. Този токен за достъп, получен от крайната точка на токена, се съхранява от клиентското приложение, което след това го представя като токен на носител с всяка заявка за API. API валидира този токен и определя дали да предостави или откаже достъп до исканите ресурси.



OpenID Connect улеснява както удостоверяването на ниво клиент, така и упълномощаването за комуникация с API. Клиентското приложение, чрез създаване на заявка за удостоверяване, пренасочва потребителя към доставчика на идентичност (IDP), където той се удостоверява. IDP, функциониращ като сървър за оторизация, предоставя както самоличност, така и токен за достъп. Токенът за самоличност проверява самоличността на потребителя, позволявайки му да влезе в клиентското приложение. Междувременно токенът за достъп се изпраща с всяка заявка за API като токен на носител, предоставящ или отказващ достъп до ресурси въз основа на валидирането му от API.

По отношение на потоците OAuth 2 и OpenID Connect е важно да се отбележи, че потокът на кода за оторизация на OpenID Connect и имплицитният поток са разширения на OAuth 2. Въпреки това OpenID Connect включва също три варианта на хибридния поток, които не съществуват в OAuth 2 и вече не се считат за най-добри практики. OAuth 2 поддържа два допълнителни потока: потокът на идентификационните данни на собственика на ресурса и потокът на идентификационните данни на клиента. Първото обикновено се избягва поради неговия екран за влизане в приложението и липсата на пренасочване, което затруднява обединяването с други IDP и сценариите за единично влизане. Последният, подходящ за комуникация машина-машина без участие на потребителя, не е част от OpenID Connect, тъй като му липсва стъпка за удостоверяване на потребителя.

Проверката на токен за достъп, който не е необходимо да бъде JWT, разкрива полета като „аудитория“ (обхват на API), „client\_id“ (идентификатор на клиентското приложение) и „обхвати“ (ресурси, достъпни с токена). Токенът също така съдържа „препратка към методите за удостоверяване“ и може да предостави достъп до информация, свързана със самоличността.



Процесът на код за оторизация протича включва използването на поток от код за оторизация с PKCE защита в уеб приложение. Започва с генерирането на произволен низ, code\_verifier, който след това се хешира, за да се създаде code\_challenge. Уеб приложението изпраща заявка за удостоверяване, съдържаща този code\_challenge, до доставчика на идентичност (IDP), където се съхранява. След удостоверяване на потребителя и незадължително съгласие, IDP пренасочва обратно към уеб приложението, включително кода за оторизация в URI.

Уеб приложението, удостоверено с client\_id и client\_secret, изпраща кода за оторизация и оригиналния code\_verifier на IDP, който хешира последния, за да провери съответствието му със съхранения code\_challenge. Ако съвпадат, IDP връща access\_token и identity\_token. identity\_token се валидира от уеб приложението, което включва хеширане на access\_token, за да се провери съответствието му със стойността at\_hash в identity token. Ако проверката е успешна, потребителят влиза в уеб приложението, като използва самоличността на твърденията, извлечена от identity\_token.

Междувременно access\_token се използва за удостоверяване на заявки към API на приложението, изпратен като токен на носител в заглавката на заявката за оторизация. API валидира access\_token в процес, различен от процедурата за валидиране на уеб приложението. При успех достъпът до ресурсите се предоставя въз основа на правилата за оторизация на API.

Ролевият контрол на достъпа (RBAC) и базираният на атрибути контрол на достъпа (ABAC) са два основни модела за управление и налагане на права за достъп в информационните системи. RBAC, както подсказва името, разчита на ролите, присвоени на отделни потребители в организацията, за да диктува правата за достъп. Всяка роля носи специфичен набор от разрешения и достъпът на потребителя до ресурси се определя от присвоената му роля. Това осигурява ефективен начин за управление и регулиране на достъпа между големи потребителски групи. Въпреки това, RBAC може да бъде по-малко гъвкав, когато се справя със сложни изисквания за контрол на достъпа поради широкия си подход, ориентиран към ролите.

За разлика от това, ABAC предлага по-подробен, гъвкав подход за контрол на достъпа чрез използване на атрибути като основа за решения за достъп. Атрибутите могат да се отнасят до потребителя (като местоположение, отдел или време на достъп), ресурса, до който се осъществява достъп, или контекста на транзакцията. След това тези атрибути се използват в предварително дефинирани правила за вземане на решения за достъп. Например, дадена политика може да гласи, че служителите от определен отдел имат достъп само до конкретни ресурси през редовното работно време от мрежата на компанията. Тази динамична природа на ABAC позволява високо персонализиран контрол на достъпа и може да се адаптира по-лесно към сложни, променящи се среди.

Докато и RBAC, и ABAC имат своите силни страни, изборът между тях често зависи от специфичните нужди и сложността на изискванията за контрол на достъпа на организацията. RBAC може да е достатъчен за организации с ясни, стабилни нужди от контрол на достъпа, докато ABAC може да е по-подходящ за организации със сложни, променящи се среди, където решенията за достъп трябва да вземат предвид различни фактори. Също така си струва да се отбележи, че RBAC и ABAC не се изключват взаимно и могат да се комбинират в хибридни модели за по-всеобхватен и гъвкав контрол на достъпа.

API често стават недостъпни след известно време поради изтичане на токените за достъп. Това е вградена функция за гарантиране на сигурността, като сървърът за самоличност по подразбиране има петминутен живот за токена за самоличност, който се използва основно за влизане в клиентското приложение. Токенът не може да бъде обработен, след като изтече, въпреки че клиентското приложение може да прилага свои собствени правила за изтичане, като плъзгащо се изтичане, за да поддържа активните потребители влезли. Токените за достъп, от друга страна, издържат по-дълго и са от съществено значение за достъп до API. Техният живот може да се контролира в класа Config на Identity Provider.

Обикновено изтекъл токен изисква потребителят да премине отново през процеса на влизане, което може да е неудобно за потребителя. Токенът за опресняване смекчава това, като извлича нов токен за достъп без намесата на потребителя. Този токен се предава на доставчика на идентификация чрез POST заявка и доставчикът на идентификация валидира токена за опресняване, като изпраща обратно нови токени за достъп и опресняване. Обхватът „offline\_access“ трябва да бъде разрешен, за да направи тази заявка.

Системата често работи със самостоятелни JWT токени за достъп, които могат да бъдат валидирани локално. Животът на тези токени обаче е трудно да се контролира, след като бъдат издадени. Референтните токени решават този проблем чрез свързване към токен, съхраняван на ниво доставчик на идентичност, предлагайки повече контрол върху живота на токена. Доставчикът на самоличност валидира тези токени и действителното им съдържание се изпраща обратно към API чрез интроспекция на токена, макар и с цената на допълнителна комуникация с всяка заявка.

Нашето изследване започна с референтни токени и техния живот, подчертавайки присъщото им предимство на по-голям контрол върху техния живот. Ние очертахме стандартния процес на отмяна, обикновено въведен чрез административен инструмент за премахване на токен от хранилището на данни, и също така признахме възможността за програмно отмяна на токени от клиентско приложение, като например по време на процес на излизане на потребител.

След това нашият фокус се измести към механиката на валидирането на токени. Подчертахме процедурите за валидиране на потока на кода за оторизация, включително валидиране на токени за самоличност и токени за достъп съответно на ниво клиент и API. Валидирането на токена за самоличност включва потвърждаване на подписа на токена, еднократното искане, твърдението на издателя, искането на аудиторията и състоянието на изтичане. Токенът за достъп, от друга страна, трябва да бъде валидиран по отношение на неговия подпис, иск на издателя, статус на изтичане и иск за аудитория, като се гарантира, че той съответства на предназначения API.

Концепцията за анулиране и валидиране на токени е важна за контрол на достъпа и поддържане на сигурността. Въпреки това, тези процедури могат да варират значително при различните доставчици на идентичност и клиенти и могат да бъдат разширени въз основа на специфични изисквания.

Ние също така обяснихме ролите на токените за опресняване като идентификационни данни за получаване на нови токени за достъп и тяхната стойност като метод за удостоверяване, докато референтните токени бяха обсъдени като идентификатори, свързани със съдържание, съхранявано на ниво доставчик на идентичност.

И накрая, проучването подчертава, че процедурите за валидиране на токени, въпреки че са от съществено значение за сигурността, често са затъмнени и не са изрично посочени от стандарта OAuth 2. Изводът е, че методите, използвани за валидиране на токени за достъп, често изискват ниво на взаимодействие или координация между сървъра за ресурси и сървъра за оторизация.

Нашето разследване завършва с очакването да проучим измерението на потребителското взаимодействие, тъй като досега сме работили с потребители в паметта. Тази прогресия ще помогне да се разбере как се управляват потребителите и как това влияе върху живота на токена и процедурите за валидиране, които обсъдихме.