(NIE) BEZPIECZEŃSTWA JWT (JSON WEB TOKEN)

JWT (JSON Web Token) to mechanizm, który jest często wykorzystywany w kontekście API webowych, ale również szerzej – z powodzeniem używany jest w aplikacjach webowych czy mobilnych. JWT możemy znaleźć w popularnych standardach, jak np. OpenID Connect, spotkamy go również czasem, korzystając z OAuth2. Znajduje zastosowanie zarówno w dużych firmach, jak i mniejszych organizacjach. Dostępnych jest wiele bibliotek obsługujących JWT, a sam standard posiada „bogate wsparcie dla mechanizmów kryptograficznych". Czy to wszystko oznacza, że JWT jest mechanizmem z natury bezpiecznym? Tę wątpliwość postaram się rozwiać w dalszej części tekstu.

Michał Sajdak

DEFINICJA

JSON Web Token to, w największym skrócie, metoda zapisu tzw. deklaracji (ang. *claims*) z wykorzystaniem notacji JSON. Jeśli chodzi o formalną definicję – warto zapoznać się ze stosownym dokumentem RFC:

> *JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure (...)*[1]

Tłumacząc prostym językiem – JWT to pewien ciąg znaków w formacie JSON[2] zakodowany w strukturę JWS (JSON Web Signature) lub JWE (JSON Web Encryption). Dodatkowo każda z tych opcji musi być zserializowana w sposób kompaktowy (ang. *compact serialization* – to jedna z dwóch serializacji wyliczanych w JWS i JWE). Najczęściej w praktyce mamy do czynienia właśnie z JWS i to ta struktura nazywana jest popularnie JWT. Z kolei „deklaracja" to najczęściej prosta para typu "klucz" : "wartość"[3]. Przykład konkretnego JWT zaprezentowano na Rysunku 1.

Widać tutaj trzy długie ciągi znaków rozdzielone kropką. Struktura całości wygląda w następujący sposób:

```
nagłówek .  payload .  podpis
```

Powyższe trzy elementy są zakodowane algorytmem BASE64URL (który wygląda bardzo podobnie jak BASE64, przy czym znak plus (+) w wynikowym ciągu zamieniany jest na minus (-), z kolei slash (/) zastępowany jest podkreśleniem (_), nie ma tutaj również standardowego paddingu BASE64, złożonego normalnie ze znaków równości (=)).

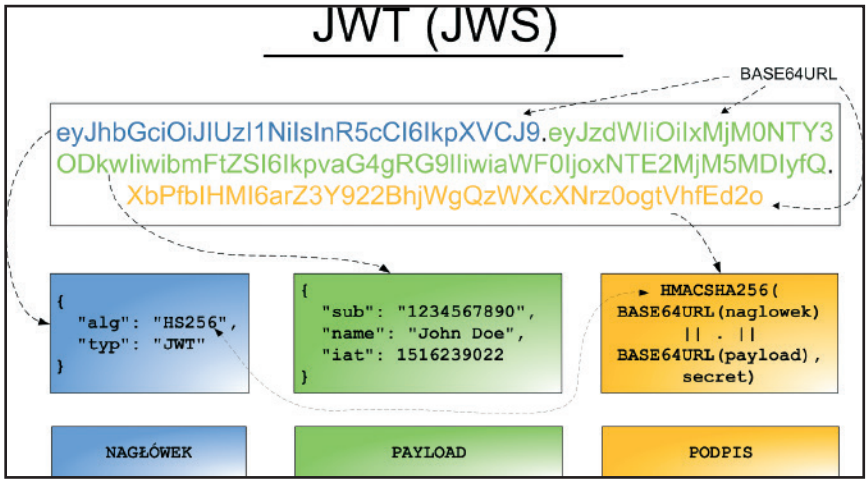
Po rozkodowaniu[4] powyższego ciągu () widzimy w pełni czytelny dla nas nagłówek, payload, a także podpis (tym razem w formie binarnej).

WAŻNE ADRESY

Celem tekstu nie jest całościowe wprowadzenie w świat JWT, jednak osobom, które chcą bardziej kompleksowo zapoznać się z tematyką, polecam następujące zasoby:

- Wprowadzenie: https://jwt.io/introduction/
- Więcej technicznych szczegółów: https://tools.ietf.org/html/rfc7519#section-4
- Najwięcej detali znajdziemy w dokumentach publikowanych przez projekt JOSE (Javascript Object Signing and Encryption – https://tools.ietf.org/wg/jose/). Poza JWT czy JWS mamy tu dodatkowo opis JWK (JSON Web Key), JWA (JSON Web Algorithms), wskazanie różnych zastosowań opisanych mechanizmów (w OAuth czy OpenID Connect); okazuje się, że JWS może mieć więcej niż jeden podpis, JWT mogą być zagnieżdżone w sobie… a to tylko parę przykładów dalszych komplikacji. Syntetyczne podsumowanie tutaj: https://www.iana.org/assignments/jose/jose.xhtml.

1. https://tools.ietf.org/html/rfc7519
2. https://www.json.org/
3. Więcej szczegółów tutaj: https://tools.ietf.org/html/rfc7519#section-4
4. Do tego celu możemy wykorzystać serwis: https://jwt.io/



Rysunek 1. Przykład podstawowego JWT

KOLEJNY PRZYKŁAD JWT I PIERWSZE PROBLEMY BEZPIECZEŃSTWA

Przejdźmy do sedna tekstu, czyli tematyki bezpieczeństwa JWT. W tym celu przywołam tekst rozważający zalety JSON Web Tokenów jako elementów zastępujących klucz API[5]. Na marginesie warto zaznaczyć, że JWT nie musi zawsze występować w połączeniu z API, jednak w praktyce to właśnie tam można go często znaleźć. Wracając do przykładu, tego typu JWT może wyglądać tak:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOiIxNDE2OTI5MDYxI
iwianRpIjoiODAyMDU3ZmY5YjViNGViN2ZiYjg4NTZiNmViMmNjNWIiLCJzY29
wZXMiOnsidXNlcnMiOnsiYWN0aW9ucyI6WyJyZWFkIiwiY3JlYXRlIl19LCJ1c
2Vyc19hcHBfbWV0YWRhdGEiOnsiYWN0aW9ucyI6WyJyZWFkIiwiY3JlYXRlIl1
9fX0.gll8YBKPLq6ZLkCPLoghaBZG_ojFLREyLQYx0l2BG3E
```

Po rozkodowaniu funkcją BASE64URL-decode otrzymujemy:

- nagłówek:

```
{
  "alg ":  "HS256 ",
  "typ ":  "JWT "
}
```

- payload:

```
{
  "iat ":  "1416929061 ",
  "jti ":  "802057ff9b5b4eb7fbb8856b6eb2cc5b ",
  "scopes ": {
    "users ": {
```

5. https://auth0.com/blog/using-json-web-tokens-as-api-keys/

```
      "actions ": [
        "read ",
        "create "
      ]
    },
     "users_app_metadata ": {
      "actions ": [
        "read ",
        "create "
      ]
    }
  }
}
```

» podpis:

```
(binarna zawartość)
```

Jak widać, dzięki takiemu „kluczowi API" (jego główna zawartość jest w payloadzie) możemy zrealizować uwierzytelnienie (mam prawo do komunikowania się z API) oraz autoryzację (widać tutaj np. akcje możliwe do wykonania przez właściciela klucza).

Od strony bezpieczeństwa można zauważyć na początek co najmniej dwa potencjalne problemy.

Pierwszy z nich to brak poufności – byliśmy w stanie łatwo rozkodować payload (i nagłówek). Czasem to nie jest problem (a wręcz zaleta), jednak kiedy wymagamy poufności samych danych przesyłanych w tokenie – jest na to sposób: szyfrowanie tokena (JWE – JSON Web Encryption).

Drugi kłopot to potencjalna możliwość nieautoryzowanego dodania kolejnej akcji przez użytkownika – np. *delete* i tym samym ominięcie autoryzacji. W tym przypadku rozwiązaniem jest standardowa możliwość podpisywania tokenów – we wcześniejszym przykładzie widzieliśmy przecież „podpis".

Wskazany w nagłówku algorytm HS256 to standardowy HMAC-SHA256 – mechanizm zapewniający integralność całej wiadomości (dzięki niemu użytkownik nie może zmienić payloadu; albo inaczej – może, jednak akceptujący token – API – wykryje to na poziomie weryfikacji podpisu, który nie będzie się zgadzał z payloadem).

Aby skonfigurować HS256, potrzebne jest wygenerowanie klucza (ciągu znaków) i umieszczenie go w konfiguracji naszego API.

Dla jasności przekazu można w dużym uproszczeniu wyobrazić sobie podpis jako:

```
SHA-256(nagłówek || payload || klucz) – gdzie || oznacza konkatenację.
```

Formalnie rzecz ujmując, jest to nieco bardziej skomplikowane: używamy algorytmu HMAC-SHA256, do którego przekazujemy klucz i wiadomość równą wynikowi:

```
BASE64URL(UTF8(JWS Protected Header)) || '.' || BASE64URL(JWS Payload)
```

Wracając do uproszczonej definicji – jeśli ktoś podmieni payload, nie jest w stanie wygenerować nowego podpisu (bo jest do niego potrzebny klucz, który znajduje się tylko w konfiguracji API).

Podsumowując: JWT wygląda na zdecydowanie bardziej elastyczny element niż klucze API – można łatwo przekazywać dowolne dane, można zapewnić ich integralność, a także – w razie potrzeby – poufność. Dodatkowo wszystkie informacje (poza kluczem), które służą do weryfikacji kogoś, kto przedstawia dany token, mogą być w samym tokenie (otrzymujemy bezstanowość i znaczną redukcję obciążenia bazy danych). Jednak, nie ma róży bez kolców.

Kolec pierwszy: nadmierna komplikacja

Jednym z zasadniczych problemów jest fakt, że JWT – biorąc pod uwagę powiązane specyfikacje – jest jednak bardzo skomplikowanym mechanizmem. JWT / JWS / JWE / JWK, mnogość algorytmów kryptograficznych, dwa różne sposoby kodowania (ang. *serialization*), możliwość kompresji, możliwości więcej niż jednego podpisu, szyfrowanie do wielu odbiorców – to tylko kilka przykładów. Komplikacja na pewno nie jest przyjacielem bezpieczeństwa i powoduje możliwość wielu pomyłek w trakcie implementacji. Widać to choćby w omówieniu kolejnego problemu.

Kolec drugi: none

Jak już wspomniałem, często zakłada się, że „właściwy" JWT to właśnie JWT z podpisem (JWS), choć zgodnie z formalną specyfikacją JWT – nie musi być on obecny. Stosowny dokument RFC[6] wskazuje tzw. „unsecured JWT" – czyli bez sygnatury. Jak taki niepodpisany token JWT wygląda? W nagłówku mamy:

```
{
  "alg ":  "none ",
  "typ ":  "JWT "
}
```

w payloadzie – to, co zazwyczaj. Sekcja podpisu jest pusta (a przynajmniej ignorowana przez przetwarzającego taki token).

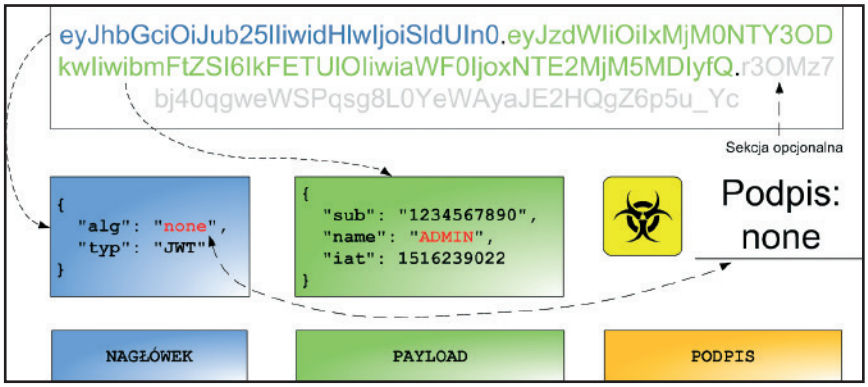
Co ciekawe, powyżej wskazany algorytm `none` to jeden z dwóch, które wg stosownego RFC muszą być zaimplementowane:

> *Of the signature and MAC algorithms specified in JSON Web Algorithms [JWA], only HMAC SHA-256 ("HS256") and "none" MUST be implemented by conforming JWT implementations.*

Co to daje atakującemu? Otóż dostajemy JWT (z podpisem), chcemy go zmienić (np. dodać sobie nowe uprawnienie) – ustawiamy więc w nagłówku `{"alg":"none"}` i dowolnie zmieniamy payload. Wysyłamy całość do API z podpisem lub bez. Czy serwer powinien przyjąć taki token? Teoretycznie tak, ale byłoby to przecież zaprzeczenie całej idei podpisów. Takie sytuacje rzeczywiście jednak miały (jeszcze mają?) miejsce w wielu bibliotekach obsługujących JWT[7]. Przykład na Rysunku 2.

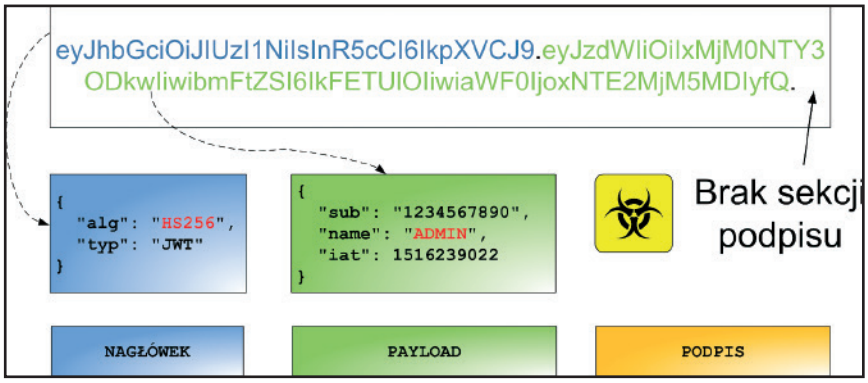
Przy okazji warto wspomnieć o innym problemie: co się stanie, jeśli w nagłówku będziemy mieli dowolny algorytm podpisu (np. HS256 czy HS512), ale z tokena usuniemy cały podpis?

6. https://tools.ietf.org/html/rfc7519
7. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/



Rysunek 2. Algorytm none

Jak widać na przykładzie opisanym w serwisie GitHub[8] (CVE-2018-1000125), czasem taki token… będzie zweryfikowany poprawnie! Można powiedzieć nieco z przekąsem – to nowoczesna odmiana problemu „none" (zob. Rysunek 3):



Rysunek 3. Brak sekcji podpisu, mimo wskazania algorytmu w nagłówku.

Czasem mamy do czynienia jeszcze z innym problemem – jeśli atakujący nie umie wytworzyć prawidłowego podpisu, to może… będzie on umieszczony w komunikacie błędu, zwracanym do samego atakującego? Nieprawdopodobne? Warto w takim razie zobaczyć tę podatność.

> *Critical Security Fix Required: You disclose the correct signature with each SignatureVerificationException.*[9]

Czyli jeśli ktoś zmienił payload i wysłał taki token do serwera, serwer nas o tym grzecznie informował, podając… poprawny token, który pasuje do naszego payloadu:

8. https://github.com/inversoft/prime-jwt/issues/2
9. https://github.com/jwt-dotnet/jwt/issues/61

```
Invalid signature. Expected S2LYALD0A20rNSqpJDWIjqFxmEUwArW8iE9HQRT5KJM= got
6A7DHMy6EV7eensz4xyVq+i0QJmn7DgMqM406XGI7Tk=
```

Żeby całość zadziałała, serwer musiał być skonfigurowany w trybie wyświetlania wyjątków do użytkownika, ale wcale nie jest to rzadka (choć niepoprawna) konfiguracja.

Kolec trzeci: łamanie hasła do HMAC

Wróćmy do podpisu, a dokładniej algorytmu HS256 (HMAC-SHA256). W jaki sposób następuje podpis? W pewnym uproszczeniu: każde miejsce, na którym działa API (np. osobne serwery), które chce mieć możliwość podpisu/weryfikacji podpisu, musi mieć ustawiony klucz[10] – np. słowo „sekret123".

Cały podpis to HMAC-SHA256 – który w pewnym uproszczeniu sprowadza się do podwójnego wykonania SHA256 na nagłówku sklejonym z payloadem i wyżej wspomnianym kluczem.[11]

Aby wytworzyć podpis dla zmienionego payloadu, należałoby znać klucz (ale jest on dostępny tylko w konfiguracji API). A może jednak byłaby możliwość jego odzyskania?

Standardowym atakiem jest tutaj użycie dowolnego tokena wygenerowanego przez API i próba łamania go – czyli klasyczny atak typu bruteforce/słownikowy/mieszany itp.

Jedna iteracja łamania wymaga wyliczenia dwóch hashy SHA256 (tak działa HMAC), a istnieją również gotowe narzędzia automatyzujące całą operację – jak choćby hashcat[12] implementujący łamanie klucza do JWT na kartach graficznych. Posiadając kilka bardzo dobrych kart graficznych, można uzyskać prędkość powyżej miliarda sprawdzeń na sekundę. Co więcej – całą operację można wykonywać zupełnie bez interakcji z API (wystarczy uzyskać jeden dowolny token z podpisem).

Przykładowa sesja łamania JWT wygląda tak:

```
Session.........: hashcat
Status..........: Running
Hash.Type.......: JWT (JSON Web Token)
Hash.Target.....: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMj...IDYRDg
Time.Started....: Fri Mar 30 16:01:35 2018 (1 min, 30 secs)
Time.Estimated..: Fri Mar 30 16:12:55 2018 (9 mins, 50 secs)
Guess.Mask......: ?1?2?2?2?2?2?2 [7]
Guess.Charset...: -1 ?l?d?u, -2 ?l?d, -3 ?l?d*!$@_, -4 Undefined
Guess.Queue.....: 7/15 (46.67%)
Speed.Dev.#1....:  198.0 MH/s (9.68ms) @ Accel:32 Loops:8 Thr:512 Vec:1
Recovered.......: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress........: 17964072960/134960504832 (13.31%)
Rejected........: 0/17964072960 (0.00%)
Restore.Point...: 0/1679616 (0.00%)
Candidates.#1...: U7veran -> a2vbj14
HWMon.Dev.#1....: Temp: 75c Fan: 48% Util: 98% Core:1873MHz Mem:3802MHz Bus:16
```

10. Terminy: klucz i hasło będą w kontekście HMAC używane zamiennie – choć w ogólności (np. w algorytmie AES) nie muszą być tożsame
11. Pełen opis HMAC można zobaczyć tutaj: https://tools.ietf.org/html/rfc2104
12. https://hashcat.net/hashcat/

Na jednej karcie Nvidia GTX 1070 uzyskujemy prędkość łamania około 200 milionów hashy na sekundę. Jeśli hashcatowi uda się złamać klucz, otrzymamy taki wynik, gdzie `secrety` to właśnie poszukiwany klucz:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkw
IiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.W7TG0x5Ed1
GLN6eOPhTNHErL8TfwBFRSQnqleolhXG0:secrety
```

Jak widać, zbyt słabe hasło ustawione w konfiguracji może prowadzić do jego odzyskania i w konsekwencji możliwości generowania przez atakującego dowolnych (przechodzących poprawnie weryfikację podpisu) tokenów (odzyskany klucz może służyć do podpisu).

Jak bardzo złożonego klucza mamy więc użyć? Odpowiedź jest zakopana w stosownym RFC (JSON Web Algorithms):

> *A key of the same size as the hash output (for instance, 256 bits for "HS256") or larger MUST be used with this algorithm. (This requirement is based on Section 5.3.4 (Security Effect of the HMAC Key) of NIST SP 800-117 [NIST.800-107], which states that the effective security strength is the minimum of the security strength of the key and two times the size of the internal hash value.).*[13]

Przy okazji warto jeszcze raz podkreślić, że łamanie hasła do JWT w ten sposób jest zupełnie niezauważalne dla części serwerowej (weryfikującej podpis) – wykrycie tego faktu może więc być bardzo trudne (token „prawdziwy" od tokena „sfałszowanego" w zasadzie nie będzie się różnił – podpisy są poprawne na obu!).

Kolec czwarty: gdzie algorytmów sześć, tam nie ma bezpieczeństwa

Większość problemów bezpieczeństwa z JWT to problemy implementacji (skomplikowanego) standardu. Zobaczmy przykład.

Jaki możemy wybrać algorytm podpisu w JWS? Do tej pory wspominałem o HMAC (i to tylko z funkcją SHA256), ale to nie jedyna opcja[14].

Częstą opcją jest użycie algorytmu asymetrycznego – RSA. W tym przypadku zobaczymy w nagłówku „alg":, „RS512" lub „alg": „RS256".

Przypomnijmy – do podpisu w RSA służy klucz prywatny, a skojarzony z nim klucz publiczny może weryfikować podpis. Zamiast więc symetrycznego klucza np. z algorytmu HS256 generujemy parę kluczy RSA.

Swoją drogą, niektórym podejrzane może się wydać oznaczenie RS512 czy RS256 – czyżby mamy tutaj wymóg stosowania 512 czy 256 bitowych kluczy RSA? Przecież tego typu klucze są obecnie łamalne minimalnym kosztem i niewielkim nakładem czasu[15]. Obecnie nawet klucz 1024-bitowy RSA nie jest uznawany za bezpieczny.

Na szczęście w powyższym oznaczeniu chodzi o funkcję SHA wykorzystywaną w kooperacji z RSA w celu realizacji do podpisu. RS512 oznacza więc RSA plus funkcja SHA512. Ale co z kluczem RSA? Tutaj o jego długości decyduje osoba go generująca, co stanowi ko-

13. https://tools.ietf.org/html/rfc7518

14. Przegląd po różnych wariantach podpisu można zobaczyć np. tutaj: https://auth0.com/blog/json-web-token-signing-algorithms-overview/

15. Porównaj np. https://eprint.iacr.org/2015/1000.pdf czy: https://www.theregister.co.uk/2010/01/07/rsa_768_broken/

lejny potencjalny problem (co więcej, na różnych forach można znaleźć konkretne polecenia wykorzystujące openssl i generujące klucze 1024-bitowe).

Przy okazji warto też pamiętać, że wykorzystanie RSA będzie miało wpływ na wydajność całego systemu (weryfikacja podpisu), czyli w tym przypadku wybieramy wariant wolniejszy niż HS256.

Wracając do sedna, przy okazji algorytmu RSA mamy jeszcze co najmniej jeden ciekawy problem bezpieczeństwa. Jak pisałem wcześniej, klucz publiczny jest używany do weryfikacji podpisu, zazwyczaj będzie on więc ustawiony w konfiguracji API jako `klucz_weryfikujący`.

Tutaj warto zwrócić uwagę na fakt, że dla HMAC mamy tylko jeden klucz symetryczny realizujący podpis i weryfikację.

Co może teraz zrobić atakujący, aby spróbować wytworzyć dowolny, prawidłowo podpisany token?

1. Zdobyć klucz publiczny (sama jego nazwa wskazuje, że… może być publiczny). Czasem jest on przesyłany w samym JWT.
2. Wysłać przygotowany przez siebie token – uwaga – z ustawionym w nagłówku algorytmem HS256 (czyli HMAC, nie RSA) i podpisać token za pomocą klucza publicznego RSA. Tak, to nie pomyłka – stosujemy klucz publiczny RSA (który podajemy w formie ciągu znaków) jako klucz symetryczny do HMAC.
3. Serwer otrzymuje token, sprawdza, jakiego algorytmu użyto do podpisu (HS256). `klucz_weryfikujący` został ustawiony w konfiguracji jako klucz publiczny RSA, więc…
4. Podpis się zgadza (bo użyto dokładnie tego samego klucza do weryfikacji, co do wytworzenia podpisu, a algorytm podpisu został ustawiony przez atakującego na HS256).

W czym tkwił tutaj problem? W tym, że realnie daliśmy możliwość podania algorytmu podpisu przez użytkownika, choć w zamierzeniach mieliśmy weryfikować podpis tokena tylko za pomocą RSA. Zatem albo wymuszamy tylko jeden wybrany algorytm podpisu (nie dajemy możliwości jego zmiany przez zmianę tokena), albo zapewnijmy osobne sposoby weryfikacji (i klucze!) dla każdego algorytmu podpisu, który obsługujemy.

Kolec piąty: choć może należałoby mu się pierwszeństwo

Jak się okazuje, z powodu błędu w bibliotece obsługującej JWT można czasem… po prostu podać w payloadzie tokena własny klucz, który następnie zostanie użyty przez API do weryfikacji tego tokena! Brzmi nieprawdopodobnie? W takim razie warto zobaczyć szczegóły podatności CVE-2018-0114 w bibliotece node-jose. Zacytuję oryginalny opis niemal w całości:

> *The vulnerability is due to node-jose following the JSON Web Signature (JWS) standard for JSON Web Tokens (JWTs). This standard specifies that a JSON Web Key (JWK) representing a public key can be embedded within the header of a JWS. This public key is then trusted for verification. An attacker could exploit this by forging valid JWS objects by removing the original signature, adding a new public key to the header, and then signing the object using the (attacker-owned) private key associated with the public key embedded in that JWS header.*[16]

16. https://tools.cisco.com/security/center/viewAlert.x?alertId=56326

Jak widać, jest tu wskazanie na problem bezpieczeństwa w samym RFC, który dotyczy podpisu JWT – klucz opcjonalnie można przekazać w formie struktury JWK (JSON Web Key) w nagłówku JWT. Generujemy więc swoją parę kluczy RSA. Publiczny dołączamy do tokena w formie JWK, a prywatnym podpisujemy token. API weryfikuje podpis… dostarczonym kluczem publicznym. Problem nie jest nowy, wcześniej (2016 rok) wykryto podobną podatność w bibliotece Go-jose:

> *RFC 7515, https://tools.ietf.org/html/rfc7515#section-4.1.3 "jwk" (JSON Web Key) Header Parameter allows the signature to include the public key that corresponds to the key used to digitally sign the JWS. This is a really dangerous option.*[17]

Kolec szósty: czy szyfrowanie JWT może w ogóle działać?

Co z szyfrowaniem (JSON Web Encryption) – tutaj do wyboru mamy kilka opcji algorytmów (szyfrowanie samej wiadomości, szyfrowanie klucza symetrycznego użytego do szyfrowania wiadomości). I ponownie – tutaj istnieją niezbyt pozytywnie nastrajające badania, a mianowicie kilka implementacji JWE umożliwiało odzyskanie klucza prywatnego przez atakującego.[18]

Dokładniej problem istniał w implementacji algorytmu ECDH-ES (który swoją drogą ma status „Recommended+" w stosownym dokumencie RFC[19]).

Może jest to wyjątek? Zobaczmy na algorytm AES z trybem GCM, ale tutaj bywa średnio:

> *GCM is fragile but its implementations were rarely checked.*[20]

Do wyboru mamy też choćby algorytm RSA with PKCS1v1.5 padding. Co jest z nim nie tak? Problemy są znane od 1998 roku[21]. A niektórzy podsumowują to tak:

> *PKCS#1v1.5 is awesome – if you're teaching a class on how to attack cryptographic protocols.*[22]

WAŻNE ADRESY

Więcej szczegółów odnośnie możliwych algorytmów i ewentualnych problemów można zobaczyć np. tutaj:

- https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid
- https://paragonie.com/blog/2016/12/everything-you-know-about-public-key-encryption-in-php-is-wrong
- https://paragonie.com/blog/2018/04/protecting-rsa-based-protocols-against-adaptive-chosen-ciphertext-attacks

Czy zawsze będziemy skazani na porażkę, stosując JWE? Oczywiście nie – warto jednak zweryfikować, czy używamy odpowiednio bezpiecznego algorytmu szyfrującego (i jego

17. https://mailarchive.ietf.org/arch/msg/jose/gQU_C_QURVuwmy-Q2qyVwPLQIcg
18. https://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html
19. https://tools.ietf.org/html/rfc7518
20. https://rwc.iacr.org/2017/Slides/nguyen.quan.pdf
21. ftp://ftp.rsa.com/pub/pdfs/bulletn7.pdf
22. https://blog.cryptographyengineering.com/2012/09/06/on-provable-security-of-tls-part-1/

bezpiecznej implementacji). To ostatnie może być trudne, sprawdźmy więc przynajmniej, czy w używanej przez nas bibliotece do obsługi JWE nie wykryto istotnych podatności w tym obszarze.

Kolec siódmy: dekodowanie/weryfikacja – co za różnica?

Początek był prosty, ale możemy czuć się przytłoczeni mnogością opcji. Chcemy w końcu tylko po stronie API „odkodować" token i użyć zawartych w nim informacji. Pamiętajmy jednak, że „odkodować" nie oznacza zawsze tego samego co „zweryfikować" – niby oczywiste, ale różne biblioteki mogą dostarczać różnych funkcji do dekodowania i/lub weryfikacji tokenów.[23]

W skrócie – jeśli użyję tylko funkcji typu `decode()` w API, to czasem wykonam tylko dekodowanie payloadu (ew. też nagłówka) z BASE64URL, bez weryfikacji.

Weryfikacja może być osobną funkcją udostępniania przez bibliotekę, choć może być wbudowana w funkcję typu `decode()`.

Czasem to wręcz użytkownicy proszą o taką opcję – w zacytowanym poniżej przypadku o przeciążenie metody `decode()`, tak żeby mogła również akceptować sam token (bez klucza):

> *I have seen a issue in the framework to get the payload of an JWT. To get a payload of an JWT without validation we don't need a key/secret. So in the file jwt/src/JWT/JwtDecoder.cs we missed a overload, for the method Decode that only need a token.*

Jeśli teraz programista korzystający z biblioteki użyje najprostszego wywołania: `decode(token)`, to podpis nie będzie weryfikowany.

Dodatkowo w cytowanej wcześniej dyskusji jest jeszcze pytanie o możliwość weryfikowania podpisu po stronie klienckiej, co jak już wiemy w przypadku HS256 wiąże się z użyciem klucza, który zazwyczaj powinien być… sekretem. Niestety często użycie JWT sprowadza się do poniższych czynności: wybierzmy pierwsze z brzegu algorytmy, przekopiujmy kawałki kodu z Internetu. Działa? Owszem – więc w czym problem?

Kolec ósmy: przechwycenie dowolnego tokena = przejęcie dostępu do API?

Jedną z często wskazywanych zalet JWT jest realizacja uwierzytelnienia (lub autoryzacji – zależy w jakim kontekście zostanie całość użyta) bez konieczności realizowania zapytania do bazy danych. Co więcej, możemy to zrobić równolegle na kilku niezależnych serwerach. W końcu sama zawartość tokena wystarcza do podjęcia tu decyzji. Ma to też pewną wadę – co w przypadku, gdy dostępny na wielu serwerach klucz podpisujący w jakiś sposób wycieknie? Będzie można oczywiście generować wtedy prawidłowo podpisane tokeny i zaakceptują je wszystkie maszyny, które do weryfikacji stosują odpowiedni klucz. Co może dzięki temu uzyskać atakujący? Na przykład nieautoryzowany dostęp do funkcji API czy kont innych użytkowników.

Mamy tu również potencjalnie inny problem – co w przypadku kiedy jeden wygenerowany token może być użyty w wielu różnych kontekstach? Na przykład generujemy prawidłowy token z zawartością `{ "login" = "manager" }` – i w jednej funkcji API daje on możliwość edycji pewnych danych; ale jednocześnie zapomnieliśmy że zupełnie inna funkcja otrzymująca ten sam token daje możliwość pełnego dostępu!

23. Przykład tego typu pytania czy wątpliwości można znaleźć tutaj: https://github.com/auth0/jwt-decode/issues/4.

W tym przypadku istnieje możliwość użycia pewnych parametrów definiowanych przez samą specyfikację, np.: `iss` (issuer) oraz `aud` (audience). Dzięki nim można generować tokeny, które mogą być następnie przyjęte tylko przez określonych naszych odbiorców.

Na przykład:

```
{
 "iss " =  "my_api ",
 "login " =  "manager ",
 "aud " =  "store_api "
}
```

Można tu wskazać jeszcze jeden problem – pewne zarezerwowane słowa kluczowe w JWT (*Registered Claim Names)* – np. iss/aud/iat/jti – mogą być umieszczone przy zupełnie dowolnych elementach definiowanych przez użytkownika JWT – np. login z naszego przykładu. To rodzi kolejne zamieszanie, które wskażę w kolejnym problemie.

Kolec dziewiąty: replay JWT

Co w przypadku, kiedy konkretny token powinien być użyty tylko raz? Tutaj wyobraźmy sobie scenariusz, gdy użytkownik zapisuje wygenerowany token umożliwiający wykonanie metody DELETE w naszym API. Następnie, np. po roku – kiedy teoretycznie już nie ma stosownych uprawnień – próbuje go użyć ponownie (tzw. atak typu *replay*).

Sposobem na to może być użycie pól: jti oraz exp. Jti (JWT ID) to identyfikator tokena, który musi być unikalny, a exp – to określenie daty ważności tokena. Połączenie obu pól da nam odpowiednio krótką ważność tokena oraz jego unikalność.

Warto jednak zwrócić uwagę na to, czy mamy poprawną implementację obsługi tych pół – tutaj polecam spojrzeć na błąd, w którym wartość exp w ogóle nie była brana pod uwagę.[24]

```
JWT not throwing ExpiredTokenException in .NetCore
```

Tworzący bibliotekę użyli do sprawdzania wartości pola Expiry (niezgodnie ze specyfikacją JWT), a błąd po zgłoszeniu został skorygowany.

Kolec dziesiąty: ataki czasowe na podpis

Jeśli podpis z JWS sprawdzany jest bajt po bajcie z podpisem, który jest prawidłowy (wygenerowanym po stronie akceptującej JWS), oraz sprawdzanie to kończy działanie na pierwszym niezgodnym bajcie, możemy być podatni na atak czasowy.

Zauważmy, że w takim przypadku im więcej bajtów zgodnych, tym więcej potrzebnych jest porównań i stąd dłuższy czas potrzebny na odpowiedź.

Można więc obserwować czasy odpowiedzi, generując kolejne podpisy – rozpoczynając od pierwszego bajtu podpisu, później przejść do drugiego itp.[25]

Według cytowanego opracowania, przy dużej ilości generowanego ruchu (aż 55 tysięcy requestów na sekundę) podpis dowolnej wiadomości można by uzyskać w 22 godziny (wa-

24. https://github.com/jwt-dotnet/jwt/issues/134

25. Dokładny opis takiego ataku można zobaczyć tutaj: https://hackernoon.com/can-timing-attack-be-a-practical-security-threat-on-jwt-signature-ba3c8340dea9

runki laboratoryjne). Przy mniejszym ruchu oczywiście będziemy potrzebowali więcej czasu (kilka/kilkanaście dni) – ale efekt może być porażający (możemy wygenerować dowolny JWT i przygotować podpis, który będzie zweryfikowany jako poprawny).

Czy atak jest rzeczywiście możliwy do zrealizowania w praktyce? Jak można sobie wyobrazić, zmiany w czasie odpowiedzi są minimalne, ale można je jednak próbować mierzyć.[26]

Udało się tutaj osiągnąć następujące pomiary:

> *Our work analyzes the limits of attacks based on accurately measuring network response times and jitter over a local network and across the Internet. We present the design of filters to significantly reduce the effects of jitter, allowing an attacker to measure events with 15-100µs accuracy across the Internet, and as good as 100ns over a local network.*

Z kolei przykład zgłoszenia konkretnej tego typu podatności można znaleźć w temacie zapoczątkowanym przez Jason Goodwin[27] oraz Malcolma Fella.[28]

Kolec jedenasty: mnogość bibliotek

Jako jeden z pierwszych problemów dotyczących JWT wymieniłem mnogość dostępnych opcji i różnorakich algorytmów. Na całość nakłada się również duża liczba często niekompletnych implementacji JWT, pisanych, delikatnie mówiąc, na kolanie. Niełatwy i czasem generujący problemy bezpieczeństwa standard, wymagane użycie skomplikowanych algorytmów kryptograficznych przez osoby często nie posiadającej głębokiej wiedzy o samej kryptografii… zgodnie z zasadą *garbage in, garbage out*: trudno spodziewać się tutaj super bezpiecznych bibliotek.

WAŻNE ADRESY

Część błędów w bibliotekach wymieniłem już wcześniej. Inne przykłady tutaj:

- https://www.cvedetails.com/cve/CVE-2017-12973/
- https://www.cvedetails.com/cve/CVE-2017-10862/
- https://pivotal.io/security/cve-2017-2773
- https://github.com/auth0/node-jsonwebtoken/issues/212
- https://github.com/auth0/node-jsonwebtoken/commit/adcfd6ae4088c838769d169f8cd9154265aa13e0

Przy okazji warto wspomnieć jeszcze o ogólnych problemach bezpieczeństwa dotyczących wykorzystywanych bibliotek. Po pierwsze, warto zastanowić się, czy używam biblioteki bez znanych publicznie podatności? A może używa ona podatnych zależności? Czy mam opracowane monitorowanie wykorzystanej biblioteki – na wypadek, że ktoś zlokalizuje w przyszłości istotną podatność?

26. W tym miejscu warto też przypomnieć nieco starszy tekst o atakach czasowych: https://www.cs.rice.edu/~dwallach/pub/crosby-timing2009.pdf.
27. https://github.com/jasongoodwin/authentikat-jwt/issues/12
28. https://github.com/emarref/jwt/pull/20

ALTERNATYWA DO JWT?

Patrząc na wiele problemów bezpieczeństwa, które wskazałem w JWT, można się zastanawiać, czy mamy jakąś sprawdzoną alternatywę? Obecnie odpowiedź jest raczej negatywna. Jednym z nowych pomysłów jest PASETO.

> *Paseto is everything you love about JOSE (JWT, JWE, JWS) without any of the many design deficits that plague the JOSE standards.*[29]

Czyli w wolnym tłumaczeniu PASETO ma być bezpieczną wersją JWT. Czy rzeczywiście spełni obietnice? Ciężko obecnie powiedzieć – jest to bardzo młody projekt, cały czas znajdujący się w fazie rozwoju (stan na początek 2018 roku).[30]

CZY JWT MOŻE BYĆ BEZPIECZNE?

Zdania w środowisku osób zajmujących się bezpieczeństwem są podzielone. Niektórzy kategorycznie odradzają użycia JWT, inni wskazują jedynie na słabo przygotowane implementacje, pozostali z kolei opisują dokładnie same mechanizmy JWT, pozostawiając decyzję użytkownikowi. Na całość nakłada się fakt, że JWT jest jednak bardzo popularnym mechanizmem, do którego nie ma ustandaryzowanej alternatywy, która dodatkowo udowodniła swoje bezpieczeństwo.

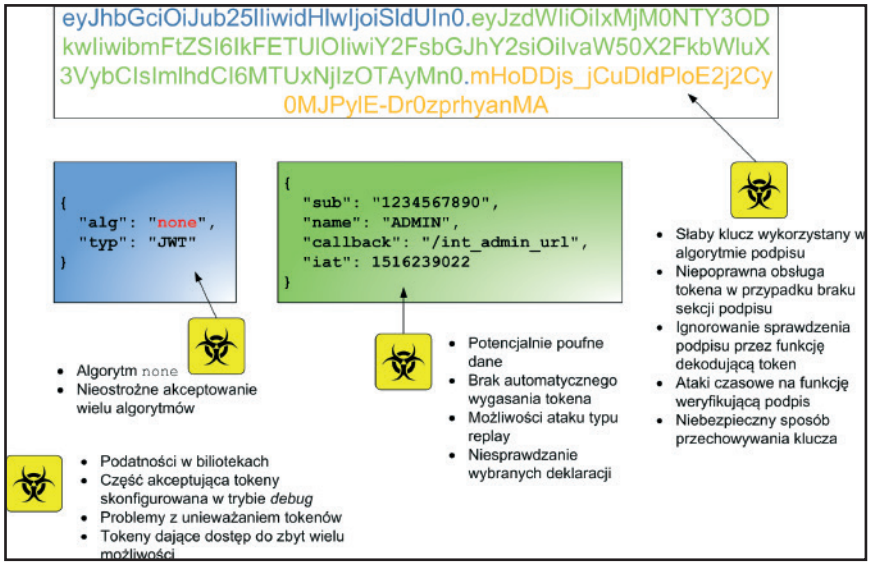
Wskazane przeze mnie wcześniej problemy bezpieczeństwa można umieścić w kilku kategoriach:

1. Problemy samej specyfikacji JWT (np. algorytm none).
2. Błędy implementacji bibliotek, w tym błędy implementacji algorytmów kryptograficznych (prawdopodobnie jest to najliczniejsza grupa problemów).
3. Niepoprawne użycie biblioteki.

Większość częstych problemów zaprezentowano na Rysunku 4.

29. https://paseto.io/

30. Więcej informacji można znaleźć tutaj:
https://paragonie.com/blog/2018/03/paseto-platform-agnostic-security-tokens-is-secure-alternative-jose-standards-jwt-etc.
Na temat motywacji do powstania projektu przeczytamy natomiast tutaj:
https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid.



Rysunek 4. Zestawienie problemów bezpieczeństwa w JWT (JWS)

Podsumujmy jeszcze całość praktycznymi radami mogącymi zwiększyć bezpieczeństwo JWT. Część zaleceń może zawierać skróty myślowe, które są jednak wyjaśnione wcześniej w tekście, dodatkowo rozszerzone linkami do materiałów źródłowych.

Na początek

1. Zrozum to, z czego chcesz skorzystać: zastanów się, czy potrzebujesz JWS czy JWE, wybierz stosowne algorytmy, zrozum zasadę ich działania (przynajmniej na ogólnym poziomie – np. HMAC, klucz publiczny, prywatny). Sprawdź dokładnie, jakie możliwości oferuje biblioteka, którą wybrałeś do obsługi JWT. Może istnieje już gotowy, prostszy mechanizm, który możesz wykorzystać?

Klucze

2. Używaj odpowiednio złożonych kluczy symetrycznych/asymetrycznych.
3. Miej przygotowany scenariusz na wypadek kompromitacji (ujawnienia) jednego z kluczy.
4. Przechowuj w odpowiednio bezpieczny sposób klucze (np. nie zapisuj ich na trwałe w kodzie źródłowym).
5. Nie pozwalaj na ustawienie algorytmu podpisu wysyłającemu token (najlepiej wymuś konkretny algorytm podpisu po stronie serwerowej).

Podpis

6. Sprawdź, czy Twoja implementacja nie akceptuje algorytmu podpisu none.
7. Sprawdź, czy Twoja implementacja nie akceptuje pustego popisu (czyli go nie sprawdza).
8. W przypadku użycia JWE sprawdź, czy używasz bezpiecznych algorytmów oraz bezpiecznej implementacji tych algorytmów.
9. Rozróżnij funkcje `verify()` od `decode()`. Innymi słowy – sprawdź, czy na pewno weryfikujesz podpis.

Ogólne zasady

10. Sprawdź, czy wygenerowany w jednym miejscu token nie może być użyty w innym w celu uzyskania nieuprawnionego dostępu.

11. Sprawdź, czy został wyłączony tryb debug i czy nie można go włączyć prostym trikiem (np. `?debug=true`).
12. Unikaj przesyłania tokenów w URL-u (nie jest to bezpieczne – np. tokeny są wtedy zapisywane bezpośrednio do logów webserwerów).

Payload

13. Sprawdź, czy w payloadzie JWS nie umieszczasz poufnych informacji.
14. Sprawdź, czy chronisz się przed ponownym wysłaniem (atak typu *replay)* tokena.
15. Sprawdź, czy tokeny mają odpowiednio krótką ważność (np. poprzez wykorzystanie deklaracji „exp"). Sprawdź, czy „exp" rzeczywiście jest poprawnie sprawdzane.
16. Zastanów się, czy potrzebujesz funkcji unieważnienia pojedynczych tokenów (sam standard na to nie pozwala, jednak istnieje kilka sposobów implementacji tego typu mechanizmu).

Biblioteki

17. Przeczytaj dokładnie dokumentację biblioteki.
18. Sprawdź podatności w wykorzystywanej przez siebie bibliotece (np. w serwisie: cvedetails.com czy na stronie projektu).
19. Sprawdź, czy Twoje poprzednie projekty nie korzystają z podatnej biblioteki; sprawdź, czy monitorujesz nowe błędy w bibliotece (mogą się one pokazać np. po miesiącu od wdrożenia).
20. Śledź nowe podatności w bibliotekach obsługujących JWT. Być może ktoś w przyszłości znajdzie podatność w innym projekcie, która w dokładnie takiej samej formie występuje w wykorzystywanym przez Ciebie rozwiązaniu.

W ramce „W sieci" znajduje się podsumowanie interesujących zasobów pokazujących zarówno problemy z JWT, jak i dobre praktyki postępowania z tym mechanizmem.

WAŻNE ADRESY

- JSON Web Token Best Current Practices:
  https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-01
- JWT Handbook:
  https://auth0.com/resources/ebooks/jwt-handbook
  (załącznik: „Best Current Practices")
- Dyskusja o słabościach JWT:
  https://lobste.rs/s/r4lv76/jwt_is_bad_standard_everyone_should_avoid
- Wybrane słabości w JWT wg OWASP:
  https://www.owasp.org/index.php/JSON_Web_Token_(JWT)_Cheat_Sheet_for_Java
- Kilka pomysłów na bezpieczne użycie JWT:
  https://dev.to/neilmadden/7-best-practices-for-json-web-tokens
- Zbiór argumentów przeciwko używaniu JWT do tworzenia sesji:
  http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/
- Porównanie JWT z identyfikatorami sesji oraz rady dotyczące odpowiednich zabezpieczeń:
  http://by.jtl.xyz/2016/06/the-unspoken-vulnerability-of-jwts.html