Uwierzytelnianie w aplikacjach webowych z użyciem kryptografii asymetrycznej

Obecnie najbardziej rozpowszechnionym schematem autoryzacji i autentykacji w Internecie jest wymaganie podania prawidłowej nazwy użytkownika i hasła. Niekiedy wprowadza się dwuskładnikowe uwierzytelnianie za pomocą fizycznego tokenu albo aplikacji w smartfonie. Celem tego artykułu jest zaprezentowanie implementacji znanych metod uwierzytelniania opartych o kryptografię asymetryczną, które do tej pory występowały wyłącznie poza aplikacjami internetowymi, a obecnie możliwa jest ich implementacja przy użyciu Web Crypto API [1].

WPROWADZENIE

Web Crypto API jest zunifikowanym interfejsem służącym do wykonywania podstawowych operacji kryptograficznych na poziomie skryptów JavaScript działających po stronie klienta. Możliwości obecnych wersji uwzględniają między innymi szyfrowanie, deszyfrowanie, generowanie kluczy, hashowanie, podpisywanie i weryfikację podpisów. Wspomniane operacje są wspierane dla wybranych algorytmów kryptografii symetrycznej (m.in. AES), asymetrycznej (m.in. RSA, ECDSA, ECDH) i/lub funkcji mieszających (m.in. SHA).

Jednym z proponowanych przez W3C przypadków użycia Web Crypto API jest „uwierzytelnianie wieloskładnikowe" [2], również w zakresie zastąpienia mało przyjaznego użytkownikom i znacznie trudniejszego w konfiguracji mechanizmu uwierzytelniania klienta na poziomie TLS.

Z punktu widzenia uwierzytelniania najbardziej istotnymi operacjami będą te związane z podpisywaniem, przy wykorzystaniu algorytmów kryptografii asymetrycznej. W takim przypadku jedna ze stron komunikacji posiada klucz prywatny (umożliwiający podpisywanie wiadomości), a druga jedynie klucz publiczny (umożliwiający weryfikację poprawności tych podpisów).

Kryptografia symetryczna i asymetryczna

Czym się różnią? Najbardziej intuicyjny będzie tutaj przykład szyfrów symetrycznych (np. AES), czyli takich, w których operacje szyfrowania i deszyfrowania są wykonywane za pomocą jednego i tego samego klucza. Zupełnie inaczej działają algorytmy szyfru asymetrycznego, albowiem najpierw konieczne jest wygenerowanie spójnej pary kluczy, gdzie jeden z nich nazywamy „publicznym", a drugi „prywatnym". Klucze te są ze sobą powiązane w taki sposób, że jedną z operacji (szyfrowanie/deszyfrowanie) da się wykonać wyłącznie kluczem prywatnym, a operację do niej odwrotną – wyłącznie kluczem publicznym.

Dlaczego w ogóle potrzebujemy algorytmów kryptografii asymetrycznej?

Największą bolączką związaną w praktyce z szyframi symetrycznymi jest konieczność uzgodnienia klucza pomiędzy rozmówcami (klientem i serwerem) w bezpieczny sposób. Czy wyobrażalne jest, żeby Google rozsyłał listy tradycyjne ze zdrapką zawierającą indywidualny klucz AES do każdego ze swoich klientów? Jest na to znacznie lepszy sposób.

Część klucza traktowaną jako publiczną można w dowolny sposób udostępniać osobom postronnym. Dzięki znajomości klucza publicznego będą one w stanie szyfrować wiadomości skierowane do jego właściciela, ale nikt inny poza nim nie będzie w stanie ich odszyfrować, nawet jeżeli je przechwyci. Dzięki takim właściwościom komunikacji możliwe jest np. uzgodnienie klucza symetrycznego w sposób uniemożliwiający jego przechwycenie.

Przykładowy schemat uwierzytelniania, przy założeniu, że komunikacja jest realizowana w sposób stanowy, może wyglądać tak:

1. Serwer generuje losowy łańcuch binarny (określoną liczbę losowych bajtów składającą się na tzw. *challenge nonce*) i przesyła go do klienta, żądając podpisania tego łańcucha.
2. Klient używa swojego klucza prywatnego, aby podpisać losowy ciąg otrzymany od serwera, i przesyła do niego swoją nazwę użytkownika oraz utworzony podpis.
3. Serwer odnajduje klucz publiczny przypisany do użytkownika o podanej nazwie i weryfikuje poprawność podpisu. Jeżeli podpis jest poprawny, to użytkownik uwierzytelnił się prawidłowo (udowodnił, że jest właścicielem klucza).

Przedstawiony scenariusz uwierzytelniania *challenge-response* będzie prawidłowy przy założeniu, że klient zweryfikował wcześniej tożsamość serwera i transmisja między nimi jest już zabezpieczona np. poprzez TLS. Bez takiego zapewnienia istnieje ryzyko, że klient połączy się ze „złośliwym serwerem", który będzie pośredniczył w komunikacji z prawdziwym serwerem, przeprowadzając atak *man-in-the-middle*.

Bardzo istotną właściwością takiego rozwiązania jest to, że serwer nie musi przechowywać żadnych wrażliwych danych związanych z uwierzytelnianiem klienta. Kłopotliwy jest np. wyciek hashowanych haseł z bazy danych na serwerze czy też wyciek współdzielonych sekretów wykorzystywanych do generowania kodów w aplikacjach mobilnych (two-factor authentication). Wyciek klucza publicznego jest stosunkowo mało kłopotliwy, bowiem jeżeli wszyscy klienci używają silnych kluczy (znacznie prostsze do zapewnienia niż silne hasła), taka informacja nie daje atakującemu żadnej znaczącej przewagi.

CO WNOSI WEB CRYPTO API DO ZABEZPIECZEŃ?

Wszystko, co zostało opisane do tej pory, może zostać zrealizowane przez dowolną bibliotekę JavaScript, która w prawidłowy sposób implementuje odpowiednie algorytmy (np. [3]). Pora na przedstawienie unikalnych korzyści wynikających z wykorzystania Web Crypto API:

SZUKASZ PRACY MARZEŃ?

↓

ZRÓB PIERWSZY KROK!

PRZYJDŹ DO SZKOŁY JĘZYKA ANGIELSKIEGO SPEAK UP

- NAUCZYMY CIĘ ANGIELSKIEGO SZYBKO I SKUTECZNIE
- STAWIAMY NA TWÓJ ROZWÓJ POPRZEZ INNOWACYJNE METODY NAUCZANIA
- GWARANTUJEMY ELASTYCZNOŚĆ I INDYWIDUALNE PODEJŚCIE
- Z NAMI ZDOBĘDZIESZ CERTYFIKAT JĘZYKOWY

THE ENGLISH SCHOOL SPEAK UP

WWW.SPEAK-UP.PL

- samodzielna implementacja algorytmów kryptograficznych jest bardzo ryzykownym i czasochłonnym zajęciem, interfejsy udostępniane przez twórców przeglądarek są znacznie bardziej wiarygodne (za sprawą doświadczenia ich developerów oraz zewnętrznych audytów) niż biblioteki w JavaScripcie;
- na poziomie JavaScriptu istnieją problemy z uzyskaniem źródła losowości nadającego się do zastosowań w kryptografii, znany `Math.random()` jest generatorem pseudolosowym, który nie nadaje się do generowania kluczy [4];
- zdecydowanie najciekawszy argument: klucze wygenerowane przez Web Crypto API da się zapisać w przeglądarce w taki sposób, aby możliwe było używanie go do generowania podpisów, ale nie było możliwości ich wyeksportowania z poziomu aplikacji.

Przejdźmy do rozwinięcia myśli z ostatniego punktu. Przeglądarki implementujące wspomniany interfejs udostępniają do JavaScriptu obiekty klasy `CryptoKey`. Rola instancji takiego obiektu w praktyce sprowadza się wyłącznie do bycia „uchwytem" na klucz, ponieważ z punktu widzenia skryptu nie zawiera ona żadnych istotnych informacji na jego temat.

Nie ma możliwości bezpośredniego odczytania liczb składających się na klucz, chociaż można to zrobić pośrednio, np. za pomocą metody `exportKey()`. Przeglądarka może jednak odmówić zrealizowania takiego żądania, jeżeli klucz nie jest oznaczony jako eksportowalny. Odmowę możemy uzyskać również przy próbie wykonania operacji z użyciem danego klucza, np. jeżeli żądamy zaszyfrowania jakiejś wiadomości, ale „szyfrowanie" nie znajduje się na liście dozwolonych sposobów użycia danego klucza.

Klucz z flagą `extractable` ustawioną na `false` można wyciągnąć jedynie z pamięci przeglądarki lub z plików bazy danych utrzymywanych przez przeglądarkę (zakładając, że klucz jest w niej zapisany). Pozwala to stwierdzić, że żaden złośliwy kod JavaScript nie jest w stanie ukraść klucza przechowywanego w ten sposób, chyba że będzie to działający exploit na silnik przeglądarki o stosunkowo wysokim poziomie krytyczności.

PERMANENTNE SKŁADOWANIE KLUCZA

Na chwilę obecną Web Crypto API nie definiuje żadnego konkretnego mechanizmu magazynowania kluczy [5]. Oczekiwanym rozwiązaniem jest użycie do tego celu bazy danych IndexedDB, która do serializacji danych wykorzystuje algorytm klonowania strukturalnego. Oznacza to, że razem z obiektami składowany jest również ich wewnętrzny stan (za wyłączeniem niektórych przypadków szczególnych), nawet jeżeli jest on niewidoczny z punktu widzenia JavaScriptu. Dzięki temu możliwe jest składowanie obiektów typu CryptoKey, nawet tych z flagą `extractable` ustawioną na `false`, flaga jest bowiem sprawdzana podczas określonych wywołań Web Crypto API i nie ma żadnego specjalnego znaczenia z punktu widzenia takiej „głębokiej" serializacji.

KOMPATYBILNOŚĆ W PRZEGLĄDARKACH

Wspomniane pomysły zadziałają na przeglądarkach, które wspierają zarówno Web Crypto API, jak i IndexedDB. Sytuacja jest o tyle dobra, że wsparcie jest obecne we wszystkich popularnych przeglądarkach (wliczając w to Internet Explorer).

W Tabeli 1 zaprezentowano przybliżone dane udostępnione w Mozilla Developer Network.

minimalna wersja	Web Crypto API (SubtleCrypto)	Indexed DB
Chrome	37	24
Edge	20	(tak)
Firefox (Gecko)	34	16.0
Internet Explorer	11 (msCrypto)	10
Opera	24	15
Safari (WebKit)	8 (webkitCrypto)	10

Tabela 1. Minimalne wersje przeglądarek obsługujące standardy Web Crypto API oraz Indexed DB

W praktyce poziom wsparcia standardu można określić jako „dość dobry". Na najnowszych wersjach przeglądarek Opera, Firefox i Chrome kod prezentowany w artykule działa bez żadnych problemów.

Przeglądarka Safari (wersje 8-10) implementuje interfejs SubtleCrypto z prefiksem `webkitSubtle`, nie jest on jednak kompletny, przykładowym problemem może być brak możliwości eksportu klucza publicznego do formatu SPKI.

Microsoft Edge oraz Internet Explorer 11 implementują wspomniany interfejs niezgodnie ze standardem. Po pierwsze, funkcja `subtleCrypto.encrypt` wymaga podania klucza `algorithm.hash` (mimo że według standardu wymagany jest jedynie `algorithm.name`). Po drugie, co odkryliśmy w trakcie pisania artykułu, próba zapisu obiektu `CryptoKeyPair` do IndexedDB powoduje błąd `DataCloneError`. Nie jest to jednak krytyczny problem, bowiem zapisywanie obiektów `CryptoKey` działa poprawnie – możliwe jest więc zapisanie „wypakowanej" ręcznie pary dwóch kluczy, co nie jest do końca intuicyjne. Znalezienie tego obejścia zajęło nam godzinę, a szczególnie dezorientujący był fakt, że wszystkie pozostałe przeglądarki nie mają problemów z zapisywaniem całych obiektów `CryptoKeyPair`. Problemy te zostały zgłoszone [6] [7] do developerów Microsoft Edge i możliwe, że niedługo zostaną rozwiązane.

Innym istotnym problemem implementacji z Internet Explorer 11 jest brak obsługi Promise, co powoduje, że cały kod odpowiedzialny za obsługę Web Crypto API musiałby zostać napisany od zera specjalnie dla IE 11 (albo należałoby użyć jakiejś warstwy pośredniej do emulacji Promise). Bolączką obu przeglądarek Microsoftu jest natomiast fakt, że w zakresie podpisów kluczem asymetrycznym wspierają one jedynie `RSASSA-PKCS1-v1_5`, który nie jest zalecany do nowych projektów. Zostanie on jednak wykorzystany w przedstawionych tutaj implementacjach dla zachowania kompatybilności z Edge. Wprowadzenie poprawki polegającej na wybieraniu rekomendowanego systemu `RSA-PSS` za każdym razem, kiedy przeglądarka użytkownika go obsługuje, nie jest trudnym zadaniem, ale zostanie tutaj pominięte w celu uproszczenia implementacji.

Wartym zauważenia jest fakt, że twórcy przeglądarek opartych na WebKit zadbali o bezpieczne wykorzystanie nowego API, albowiem `SubtleCrypto` nie jest dostępne, jeżeli strona nie została

Ostrożnie!

Chociaż przedstawione tutaj implementacje są zgodne ze sztuką, autorzy nie udzielają żadnych gwarancji związanych z bezpieczeństwem. Znajdują się one w artykule, aby umożliwić czytelnikowi lepsze zrozumienie omawianego tematu oraz samodzielne eksperymenty z kodem. Przed zastosowaniem zaproponowanych rozwiązań w aplikacji działającej w środowisku produkcyjnym konieczne jest sprawdzenie, czy prezentowane metody są odpowiednie dla danego przypadku, oraz zlecenie zewnętrznego audytu, który sprawdzi, czy zostały wdrożone prawidłowo.

załadowana przy użyciu bezpiecznego kanału komunikacji (wyjątkiem od tej reguły są aplikacje działające lokalnie).

LOGOWANIE W SCHEMACIE CHALLENGE-RESPONSE (FLASK)

Przejdźmy do implementacji przykładowego schematu *challenge-response* omówionego w sekcji „Wprowadzenie". Pełny kod źródłowy poniższego demo dostępny jest w [8], a poniżej omówione zostaną jedynie najciekawsze fragmenty.

Na początku pracy z Web Crypto API przydatne będzie opakowanie oryginalnych wywołań do postaci bardziej wysokopoziomowych funkcji, zgodnie z naszymi potrzebami.

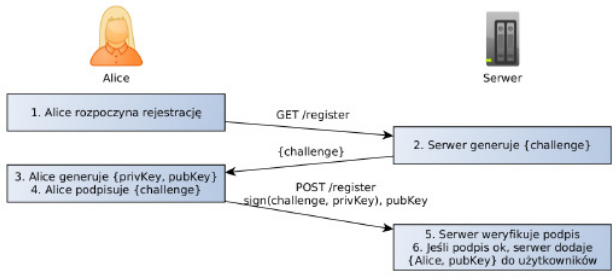
Listing 1. Podstawowe funkcje opakowujące wywołania Web Crypto API, którymi się posłużymy

```
function makeKeys() {
  var options = {
    name: "RSASSA-PKCS1-v1_5",
    modulusLength: 2048,
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
    hash: {name: "SHA-256"}
  };

  var usages = ["sign", "verify"];

  return window.crypto.subtle.generateKey(options, false, usages);
}

function signData(data, privateKey) {
  var options = {
    name: "RSASSA-PKCS1-v1_5",
    hash: {name: "SHA-256"}
  };

  return window.crypto.subtle.sign(options, privateKey, data);
}
```

Zgodnie ze swoimi nazwami funkcje `makeKeys` oraz `signData` zajmują się odpowiednio generowaniem pary kluczy RSA i podpisywaniem przekazywanej wiadomości za pomocą klucza prywatnego. Parametry kryptosystemu są dość standardowe – wykorzystywane są 2048 bitowe klucze, zwyczajowa wartość `publicExponent`: `0x10001` i dobrze znany algorytm hashujący `SHA-256`.

Przejdźmy teraz do obsługi procesu rejestracji, który opiera się na standardowym formularzu HTML. Serwer przesyła użytkownikowi formularz zawierający pola: `username` (do ręcznego wypełnienia przez użytkownika), `challenge` (wypełnione przez serwer w chwili generowania formularza), `public_key` oraz `signature` (które muszą zostać wypełnione przez JavaScript po stronie klienta).

Serwer oczekuje od klienta wygenerowania nowej pary kluczy, umieszczenia klucza publicznego zakodowanego w formacie PEM w polu `public_key` formularza oraz złożenia podpisu pod wiadomością stanowiącą złączenie klucza publicznego i wartości `challenge`. W ten sposób użytkownik podaje do wiadomości



Rysunek 1. Komunikacja klient-serwer podczas przeprowadzania procedury rejestracji

serwerowi swój klucz publiczny oraz udowadnia, że faktycznie jest dysponentem tego klucza (nie może użyć cudzego klucza publicznego, ponieważ bez klucza prywatnego nie byłby w stanie złożyć poprawnego podpisu). Podobna struktura, nazywana SPKAC (Signed Public Key and Challenge), była niegdyś produkcyjnie wykorzystywana w przeglądarkach, nie jest ona jednak binarnie równoważna z używaną tutaj strukturą.

Listing 2. Skrypt odpowiedzialny za uzupełnianie pól w formularzu rejestracji

```
function performRegister(key_label, challenge) {
  if (typeof challenge === "string") {
    challenge = base64ToBinary(challenge);
  }

  return makeKeys().then(function (keys) {
    return Promise.all([keys, exportKey('spki', keys.
    publicKey)]);
  }).then(function (vals) {
    var keys = vals[0];
    var exportedKey = vals[1];

    var publicKeyPEM = spkiToPEM(exportedKey);
    var pkac = new Uint8Array(exportedKey.byteLength +
    challenge.byteLength);
    pkac.set(new Uint8Array(exportedKey), 0);
    pkac.set(new Uint8Array(challenge), exportedKey.byteLength);

    // sign data (so we have a signed "PKAC")
    return Promise.all([keys, publicKeyPEM, signData(pkac, keys.
    privateKey)]);
  }).then(function (vals) {
    var keys = vals[0];

    // securely store the new key pair in the IndexedDB
    var dbPromise = callIndexedDB(function (store) {
      return new Promise(function (resolve, reject) {
        store.put({id: key_label, publicKey: keys.publicKey,
        privateKey: keys.privateKey});
        resolve();
      });
    });

    vals.push(dbPromise);
    return Promise.all(vals);
  }).then(function (vals) {
    var keys = vals[0];
    var publicKeyPEM = vals[1];
    var signature = binaryToBase64(vals[2]);

    return {"publicKey": publicKeyPEM, "signature": signature};
  });
}
```

Finalnie klucze zostają zapisane w bazie IndexedDB, aby możliwe było ich ponowne użycie. Klucze podlegają ochronie opisanej we wcześniejszych punktach.

Po stronie serwera obsługa formularza rejestracji wygląda następująco:

Listing 3. Obsługa formularza rejestracji po stronie serwera

```
@signature_login_poc.route('/register/submit',
methods=['POST'])
def register_submit():
  public_key_pem = request.form.get('public_key')

  try:
    public_key = load_public_key(public_key_pem.encode('ascii'))
  except MalformedPublicKey as e:
    flash('malformed public key: {}'.format(e), 'danger')
    return redirect(url_for('main'))

  try:
    challenge = session.pop('challenge_register')
  except KeyError:
    flash('no challenge was stored in session', 'danger')
    return redirect(url_for('main'))
```

```
try:
  verify_spkac(public_key, challenge, request.form.
  get('signature'))
except InvalidSignature as e:
  flash('signature verification failed', 'danger')
  return redirect(url_for('main'))

username = request.form.get('username')

if len(username) == 0:
  flash('empty username not allowed', 'danger')
  return redirect(url_for('main'))

if app.extensions['public_keys'].fetch_user_pem(username):
  flash('such user already exists', 'danger')
  return redirect(url_for('main'))

app.extensions['public_keys'].store_user_pem(username,
public_key_pem)

flash('succesfully registered as {}'.format(username),
'success')
return redirect(url_for('main'))
```

Przy czym `app.extensions['public_keys']` to generyczne repozytorium, które potrafi zapisywać/wczytywać klucz publiczny przypisany do danej nazwy użytkownika. W tej konkretnej implementacji jest to warstwa abstrakcji nad Redisem, ale nic nie stoi na przeszkodzie, by używać do tego celu bazy danych albo jakiegokolwiek innego trwałego medium.

W bardzo podobny sposób zaimplementować można proces logowania. Serwer generuje losowy ciąg `challenge`, który klient podpisuje za pomocą swojego klucza, a potem uzupełnia odpowiednie pole w formularzu wygenerowanym podpisem.

Listing 4. Skrypt odpowiedzialny za wypełnianie formularza logowania

```
function performLogin(key_label, challenge) {
  if (typeof challenge === "string") {
    challenge = base64ToBinary(challenge);
  }

  return callIndexedDB(function (store) {
    return new Promise(function (resolve, reject) {
      var getData = store.get(key_label);
      getData.onsuccess = function () {
        // we've fetched our key pair (which was generated during
        registration) from IndexedDB
        if (!getData.result) {
          reject(new NoSuchKeyPair(key_label));
        }

        resolve(getData.result);
      };
      getData.onerror = function (err) {
        reject(err);
      };
    });
  }).then(function (dbResult) {
    var exportKeyPromise = exportKey('spki', dbResult.
    publicKey);
    var signDataPromise = signData(challenge, dbResult.
    privateKey);

    return Promise.all([exportKeyPromise, signDataPromise]);
  }).then(function (vals) {
    var exportedKey = vals[0];
    var rawSignature = vals[1];

    // here we export public key only in order to display it
    // to the user it's not required in the login process,
    // as the server needs to know public key already
    var publicKey = spkiToPEM(exportedKey);

    // fill the "signature" form field with the signature of the
    // challenge it will be sent over to the server for verification
    var signature = binaryToBase64(rawSignature);

    return {"publicKey": publicKey, "signature": signature};
  });
}
```

Walidacja po stronie serwera polega na wczytaniu wartości `challenge` przesłanej do klienta oraz wcześniej zapisanego klucza publicznego przypisanego do danej nazwy użytkownika oraz na zweryfikowaniu poprawności podpisu.

Listing 5. Obsługa formularza logowania po stronie serwera

```
@signature_login_poc.route('/login/submit', methods=['POST'])
def login_submit():
  username = request.form.get('username')
  user_public_key_pem = app.extensions['public_keys'].
  fetch_user_pem(username)

  try:
    challenge = session.pop('challenge_login')
  except KeyError:
    flash('no challenge was stored in session', 'danger')
    return redirect(url_for('main'))

  if not user_public_key_pem:
    flash('no such user {}'.format(username), 'danger')
    return redirect(url_for('main'))

  user_public_key = load_public_key(user_public_key_pem)

  try:
    verify_challenge(user_public_key, challenge, request.form.
    get('signature'))
  except InvalidSignature:
    flash('you are not {}'.format(username), 'danger')
    return redirect(url_for('main'))

  flash('hello user {}'.format(username), 'success')
  return redirect(url_for('main'))
```

Jeżeli `challenge` jest podpisany poprawnie, to klientowi udało się udowodnić, że jest dysponentem klucza, co oznacza prawidłowe uwierzytelnienie (w przypadku modelowym wygenerowanie prawidłowego podpisu bez znajomości klucza prywatnego jest niemożliwe).

INTERCEPTOR W ANGULAR.JS PODPISUJĄCY KAŻDE ZAPYTANIE

Poniżej omówimy nieco bardziej zaawansowany *proof of concept* polegający na rozszerzeniu aplikacji (konkretnie przykładowej todo listy z [9]) w Angularze w taki sposób, aby każde zapytanie wychodzące od klienta było transparentnie podpisywane kluczem RSA zapisanym w przeglądarce. Pełne źródło znajduje się pod adresem [10], a poniżej zaprezentowane są jedynie wybrane elementy.

Jeden z ciekawszych fragmentów kodu to oczywiście automatyczne podpisywanie każdego wykonywanego zapytania do API. Zostało to zaimplementowane za pomocą funkcji HTTP Interceptorów w AngularJS [11] [12]:

Listing 6. Transparentne podpisywanie wszystkich żądań we frameworku AngujarJS

```
.factory('secureAuthInterceptor', function (secureSign, crypto,
$httpParamSerializer) {
  return {
    request: function (config) {
      var data = config['method'] + ' ' + config['url'] + '\n';
      if ('data' in config) {
        config['data'] = angular.toJson(config['data']);
        data += config['data'];
      }

      return secureSign.sign(data).then(function(signature) {
        var signatureHex = crypto.stringToHex(crypto.arrayBufferT
        oString(signature));
        config.headers['X-Signature'] = signatureHex;
        return config;
      });
    }
  };
});
```

W tym przypadku najpierw w jeden napis łączona jest metoda, URL oraz surowe dane zapytania (jeśli są podane), a następnie jest podpisywana za pomocą modułu `secureSign`. Podpis jest dołączany jako nagłówek `X-Signature` do zapytania i powinien być weryfikowany po stronie serwera.

Samo podpisywanie dla wygody zostało wydzielone do osobnego serwisu – dzięki temu można będzie łatwo zmienić metodę podpisywania, nie ruszając reszty kodu, oraz wykorzystać podpisywanie w innych miejscach aplikacji:

Listing 7. Serwis służący do podpisywania wiadomości

```
.factory('secureSign', function (crypto) {
  return {
    sign: function (data) {
      var keysPromise = crypto.makeKeys();
      return keysPromise.then(function(keys) {
        data = crypto.stringToArrayBuffer(data);
        return crypto.signData(data, keys.privateKey);
      });
    }
  };
})
```

Samo podpisywanie jest analogiczne jak w poprzedniej demonstracji, więc nie będzie tutaj replikowane – chętnych zapraszamy do samodzielnego zapoznania się z kodem.

Elegancją tego rozwiązania jest kompletna przeźroczystość podpisów dla reszty aplikacji. Na przykład fragment serwisu odpowiedzialnego za rozmowę z API:

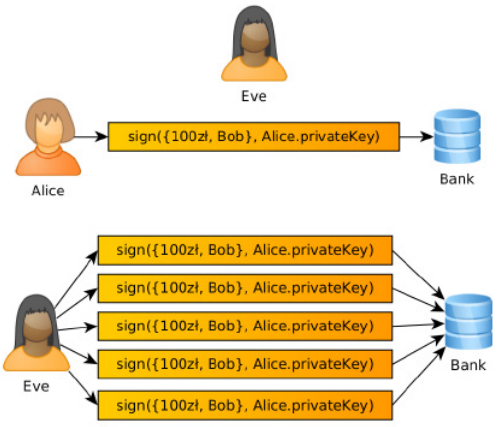
Listing 8. Serwis odpowiedzialny za rozmowę z API – zupełnie niezmieniony przez fakt, że wiadomości są podpisywane i weryfikowane w tle

```
var store = {
  todos: [],

  api: $resource('/api/todos/:id', null, {
    update: { method:'PUT' }
  }),

  delete: function (todo) {
    var originalTodos = store.todos.slice(0);

      store.todos.splice(store.todos.indexOf(todo), 1);
      return store.api.delete({ id: todo.id }, function () {
      }, function error() {
        angular.copy(originalTodos, store.todos);
    });
  },

  get: function () {
    return store.api.query(function (resp) {
      angular.copy(resp, store.todos);
    });
  },

  // ...
};
```

Jak widać, podpisywanie żądań zupełnie nie wpłynęło na resztę aplikacji. Można je spokojnie wdrożyć do już działającej strony przy minimalnej ilości przeróbek.

Czy ta metoda ma jakieś słabości? W „naiwnej" implementacji, jaką tu przedstawiamy, okazuje się, że tak. Jeśli atakującemu udałoby się przechwycić wysyłane zapytanie razem z podpisem, będzie mógł „ponawiać" zapytanie w nieskończoność.

Na przykład można sobie wyobrazić sytuację, kiedy przysłowiowa Alice chce przelać 100 zł do Boba. Wysyła więc podpisane żądanie do banku – bank jest pewny, że to Alice go wysłała, bo podpis się zgadza. Nawet jeśli przysłowiowa złośliwa Eve przechwyci w locie żądanie, nie będzie mogła zmodyfikować go w żaden sposób bez uszkodzenia podpisu. Niestety, jest możliwy inny atak – Eve może wysłać niezmodyfikowane żądanie do banku np. 10 razy – podpis będzie poprawny, więc wykonanych zostanie 10 przelewów, łącznie na kwotę 1000 zł.



Rysunek 2. Demonstracja ataku powtórzenia wykonywanego przez osobę trzecią

Taki atak jest nazywany atakiem powtórzenia (ang. *Replay Attack* [13]) i jest znanym problemem w kryptografii. W praktyce rozwiązuje się go, wprowadzając zmienny element do wiadomości, zazwyczaj na jeden z trzech głównych sposobów:

- Jednorazowe tokeny: serwer generuje losowy token, zapamiętuje go i wysyła go do Alice. Alice uwzględnia token w wiadomości, którą podpisuje (np. umieszcza token na końcu danych). Próby ponowienia zapytania nie powiodą się, bo serwer za każdym razem oczekuje innego tokenu. Problemem jest tu konieczność otrzymania tokenu od serwera przed każdym żądaniem.
- Nonce: każda wiadomość ma unikalny numer, który nigdy się nie powtarza. Zazwyczaj stosowane są tu po prostu liczby naturalne. Alice pierwsze żądanie podpisuje, doklejając „1" gdzieś do wiadomości, drugie doklejając „2" itd. Próby ponowienia zapytania nie powiodą się, bo serwer wykryje powtórzone nonce. Problemem jest tu konieczność trzymania ostatnio użytego nonce zarówno po stronie klienta, jak i serwera. Dodatkowo znacznie komplikuje sprawę wielowątkowe wysyłanie żądań.
- Timestampy: półśrodek będący uproszczoną (i mniej bezpieczną) wersją poprzednich dwóch metod. Alice podpisuje timestamp z każdą wiadomością, a serwer akceptuje żądanie, tylko jeśli timestamp jest zgodny (z pewną tolerancją). Zaleta tego rozwiązania to zero dodatkowych danych do synchronizacji – jeśli jednak atakujący odpowiednio szybko powtórzy zapytanie, atak powtórzenia ciągle może mu się udać.

Jak poradziliśmy sobie z tym atakiem w naszej przykładowej implementacji?

Odpowiedź jest prosta – wcale. Decyzja, jak podejść do tego problemu, jest bardzo specyficzna dla aplikacji, każda metoda ma swoje zalety i wady – trudno mówić o jedynym słusznym rozwiązaniu.

W praktyce często nie trzeba się tym przejmować. W przypadku ryzykownych akcji w aplikacjach webowych i tak od zawsze zalecane jest używanie tokenów CSRF. Jeśli użyty zgodnie ze sztuką token CSRF jest uwzględniony w podpisie, ochrona przed atakami powtórzenia przychodzi w pewnym sensie za darmo.

Innym, dobrym podejściem w przypadku aplikacji webowych jest często... zupełne zignorowanie problemu. Przy założeniu, że TLS nie zostanie w jakiś sposób złamany albo ominięty przez atakującego, ataki tego typu są niewykonalne. Warto jednak pamiętać o tym ataku przy przenoszeniu rozwiązań na inne płaszczyzny.

WŁAŚCIWOŚCI TEGO TYPU ZABEZPIECZEŃ

Załóżmy, że serwer z aplikacją do zarządzania akcjami giełdowymi został pomyślnie zaatakowany i osoba trzecia uzyskała możliwość wykonywania arbitralnych odczytów z bazy danych (np. w wyniku pomyślnego ataku SQL injection). Serwer przechowuje jednak wyłącznie klucze publiczne użytkowników, identyfikatory sesji oraz dane biznesowe związane z giełdą. Atakujący na podstawie tych danych nie jest w stanie zalogować się na żadnego z użytkowników ani autoryzować żadnej operacji, ponieważ nadal nie dysponuje on danymi użytecznymi do wygenerowania prawidłowego podpisu. Ponadto po wykryciu tego typu ataku nie jest konieczne wymuszanie na użytkownikach zmiany kluczy.

Jednym z potencjalnych problemów takiego rozwiązania jest słaba przenośność pomiędzy komputerami. Klucz jest przywiązany do konkretnej przeglądarki zainstalowanej na konkretnym komputerze. Jeżeli użytkownik chciałby skorzystać z aplikacji gdzieś indziej, musiałby ponownie przejść proces generowania klucza, a co za tym idzie, najpierw uwierzytelnić się w jakiś inny wiarygodny sposób oferowany przez aplikację (np. kodem z SMSa).

DODATKOWE TECHNOLOGIE ZWIĘKSZAJĄCE BEZPIECZEŃSTWO

Wcześniej wspomniany został problem związany z ryzykiem ataku XSS lub podmianą skryptu JavaScript bezpośrednio na serwerze na złośliwy skrypt. Taki skrypt nie jest w stanie wykraść klucza użytkownika, co zostało uzasadnione w poprzednich punktach, ale nadal możliwe jest przeprowadzenie specyficznego rodzaju ataku phishingowego. Taka sytuacja zawsze jest krytyczna, niezależnie od wykorzystywanego sposobu uwierzytelniania. Dodatkową warstwę ochronną może stanowić Content-Security-Policy (CSP), który został omówiony między innymi w poprzednich wydaniach „Programisty". [14]

PODSUMOWANIE

W artykule przedstawiono implementację uwierzytelniania opartą na podpisach kluczami RSA oraz wybrane problemy bezpieczeństwa powiązane z tematem. Zaprezentowany *proof of concept* pokazuje, że standard Web Crypto API może pomyślnie zostać wykorzystany do wprowadzenia dodatkowej warstwy zabezpieczeń o unikalnych właściwościach. W środowiskach wymagających mocnego uwierzytelniania może sprawdzić się dobrze w kombinacji z innymi znanymi sposobami, takimi jak uwierzytelnianie hasłem, tokenem z aplikacji czy potwierdzenie kodem SMS. Można go również z powodzeniem wykorzystać do zastąpienia tzw. ciasteczek „zapamiętaj mnie".

Inny interesujący interfejs powiązany z tematyką artykułu – Web Authentication API [17] – znajduje się obecnie w procesie standaryzacji, a jego potencjalna implementacja w przeglądarkach spowoduje znaczne zwiększenie możliwości związanych z uwierzytelnianiem i autoryzacją, ponad to, co zostało tutaj zaprezentowane.

Rozwiązania sprzętowe

Przedstawiony tutaj schemat uwierzytelniania można zrealizować również w oparciu o tokeny sprzętowe. Wówczas największą różnicą jest to, że zamiast bezpośrednio odpytywać przeglądarkę o wygenerowanie pary kluczy lub złożenie podpisu, za pośrednictwem odpowiedniego API odpytujemy o to samo token sprzętowy.

Najbardziej przyszłościowym standardem dla użytkowników końcowych jest Universal 2nd Factor (U2F), opisujący sprzęt i protokół dla tokenów USB i NFC. Standard ten jest otwarty, istnieją więc zarówno implementacje komercyjne, np. FIDO U2F od Yubico [15], jak i społecznościowe, takie jak U2F Zero [16].

Na chwilę obecną standard U2F obsługują najnowsze wersje Chrome i Opera, ale inni wiodący producenci przeglądarek również pracują nad jego wsparciem.

W sieci

[1] https://www.w3.org/TR/WebCryptoAPI/
[2] https://www.w3.org/TR/WebCryptoAPI/#multifactor-authentication
[3] https://kjur.github.io/jsrsasign/
[4] https://dl.packetstormsecurity.net/papers/general/Google_Chrome_3.0_Beta_Math.random_vulnerability.pdf
[5] https://www.w3.org/TR/WebCryptoAPI/#concepts-key-storage
[6] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/12782255/
[7] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/12782429/
[8] https://github.com/icedevml/webcrypto-rsa-login
[9] http://todomvc.com/examples/angularjs/
[10] https://github.com/msm-code/AngularJsSigningDemo
[11] https://docs.angularjs.org/api/ng/service/$http#interceptors
[12] https://thinkster.io/interceptors
[13] https://en.wikipedia.org/wiki/Replay_attack
[14] https://programistamag.pl/w-odpowiedzi-na-ataki-xss-mechanizm-content-security-policy/
[15] https://www.yubico.com/solutions/fido-u2f/
[16] https://u2fzero.com/
[17] https://www.w3.org/TR/webauthn/

MICHAŁ LESZCZYŃSKI

Programista i administrator systemowy związany z webdevem od kilku lat. Obecnie zajmuje się integracją, rozwojem i zabezpieczeniami aplikacji z branży hotelowej. Nieustannie zafascynowany tym, jak łatwo zepsuć Internet, pracuje nad nowymi rozwiązaniami w zakresie bezpieczeństwa dla swoich klientów.

JAROSŁAW JEDYNAK

Na co dzień broni bezpieczeństwa polskiego Internetu, pracując jako Security Engineer w Cert Polska. Do jego specjalności należy niskopoziomowa analiza malware oraz inżynieria wsteczna protokołów sieciowych złośliwego oprogramowania. W wolnym czasie lubi programować, pomaga w administracji serwisem 4programmers.net i nałogowo gra w konkursy CTF.