Architektura PrivMX

Łukasz Dudziński, Kamil Kaproń, Paweł Łazarski, Robert Łucarz, Maciej Muszytowski, Sebastian Smyczyński [1]*

Streszczenie
Architektura PrivMX opracowywana jest jako narzędzie tworzenia zdecentralizowanych aplikacji klient-serwer wykorzystujących szyfrowanie danych po stronie klienta. Dokument przedstawia podstawowe założenia architektury oraz ogólny opis działania jej kluczowych komponentów. Opisuje niezależnie zabezpieczony kanał transmisji danych będący wariantem protokołu TLS, najważniejsze funkcje API serwera, rozwiązania wykorzystane w standardowej bibliotece klienckiej oraz niezależną infrastrukturę PKI publikowania i weryfikacji kluczy publicznych.

Słowa kluczowe
Klient-serwer — Szyfrowanie end-to-end — Serwery zero-knowledge — Przesyłanie i przechowywanie danych — Kryptografia — ECC — Klucze jako identyfikatory — PKI — CONIKS — Drzewa Merkle — Struktury append-only

[1] *Simplito Sp. z o.o., Software R&D Dep, Toruń, Polska*
***Kontakt z autorami**: www.simplito.com/contact

Spis treści

1. Informacje ogólne

Architektura PrivMX to zestaw specyfikacji i komponentów służących do tworzenia systemów klient-serwer opartych na koncepcji szyfrowania po stronie klienta (client-side encryption) oraz serwerów z zerową wiedzą (zero-knowledge servers).

Systemy tworzone w oparciu o PrivMX pozwalają na bezpieczne przesyłanie, przechowywanie i współdzielenie danych.

Celem tego dokumentu jest ogólne przedstawienie głównych elementów Architektury PrivMX:

- szyfrowanie i odszyfrowywanie danych po stronie klienckiej i tym samym ograniczenie dostępu do danych dla serwera oraz zmniejszenie jego złożoności.
- Kontrola praw dostępu do danych oparta na metodach kryptograficznych.
- Praca w schemacie request-response i możliwość wykorzystania różnych protokołów transportowych, w tym również protokołów non-duplex takich, jak np. HTTP.
- Klient i serwer PrivMX zabezpieczają samodzielnie wzajemną komunikację, niezależnie od zabezpieczeń oferowanych przez protokół transportowy.
- Własna infrastruktura przechowywania i weryfikacji kluczy publicznych (PKI) niezależna od urzędów certyfikacji prowadzonych przez firmy trzecie.
- Możliwość budowy systemów zdecentralizowanych.

- Wprowadzenie adresacji pozwalającej na identyfikację użytkowników w systemach zdecentralizowanych.
- Możliwość udostępniania danych poprzez "dystrybucję kluczy dostępu do określonych danych" zamiast "dystrybucji (praw) użytkowników".

W zależności od konkretnych zastosowań i wymagań, system stosujący architekturę PrivMX może współpracować z dowolnymi innymi komponentami softwareowymi, również takimi, które nie szyfrują danych i pracują po stronie serwera.

W PrivMX odnaleźć można idee pochodzące z różnych technologii oraz wykorzystanie znanych i sprawdzonych algorytmów:

- SHA256, PBKDF2, HMAC-SHA512;
- ECC (Elliptic Curve Cryptography) - szyfrowanie asymetryczne - tworzenie kluczy prywatnych i publicznych oraz związane z ECC algorytmy i schematy ECDSA, ECDH(E), ECIES;
- AES (Advanced Encryption Scheme) - szyfrowanie symetryczne;
- związane z Bitcoinem – struktura Blockchain, adresy Bitcoin, BIP-32 (rozszerzone klucze i ich derywacja, chaincodes), BIP-39 (Mnemoniki do generowania kluczy)
- CONIKS (Key Transparency);
- TLS (Transport Layer Security) – procedury handshake, tickets, frames;
- SRP (Secure Remote Passwords);
- OpenPGP (RFC 4880) – pakiety PGP.

2. Komunikacja klient-serwer

2.1 Adresy PrivMX

Architektura PrivMX zakłada, iż serwery identyfikowane są poprzez hostname - nazwę domeny, a użytkownicy poszczególnych serwerów - poprzez adresy postaci user#hostname.

2.2 Procedura Service Discovery

Wywoływanie metod API serwera PrivMX możliwe jest po uzyskaniu dokładnego adresu do wysyłania żądań (API endpoint) dla danego hostname. Zajmuje się tym Procedura Service Discovery (SD), która próbuje nawiązać połączenie z serwerem docelowym hostname i odpytuje go o konfigurację dla wybranego protokołu transportowego. Przykładowo, host "simplito.com" dla protokołu "http" może zaraportować konfigurację:

```
defaultEndpoint=
  http://s1.simplito.com:3333/pmx/server
```

Konfiguracja taka, oprócz defaultEndpoint, zawierać może również inne pola, np TTL (time-to-live, czyli maksymalny czas przechowywania adresu endpointa w pamięci podręcznej), czy Redirect (nakaz szukania adresu endpointa na innym serwerze).

Procedura SD poszukując konfiguracji sprawdza na docelowym serwerze ustalone miejsca - odpytuje określone adresy URL (np. http://hostname/privmx/privmx-configuration.json) oraz sprawdza rekordy TXT we wpisie DNS dla domeny hostname. Przykładowy wpis w DNS może mieć postać:

```
V=privmx;t=http;ttl=20000;
  defaultEndpoint=
  http://hostname/privmx/server/api
```

Po wykonaniu procedury SD i uzyskaniu adresu "defaultEndpoint", moduł kliencki jest gotowy do nawiązania z serwerem połączenia przy pomocy protokołu PrivMX TLS.

2.3 Protokół PrivMX TLS

Wywołania API realizowane są przy pomocy bezpiecznego protokołu komunikacyjnego PrivMX TLS będącego zmodyfikowanym i uproszczonym wariantem protokołu TLS 1.2.

PrivMX TLS dziedziczy po TLS ogólny sposób realizacji protokołu w tym m.in. standardowy podział na ramki (CHANGE_CIPHER_SPEC, ALERT, HANDSHAKE, APPLICATION_DATA) oraz procedury handshake.

Protokół PrivMX TLS, oprócz tworzenia i utrzymywania sesji połączenia oraz szyfrowania danych przesyłanych w obie strony, dostarcza również mechanizmu logowania użytkowników i kontroli sesji zalogowanych użytkowników. Cechy charakterystyczne protokołu PrivMX TLS, to:

- działanie na dowolnym transporcie (również non-duplex), w schemacie request-response.
- Kontrola sesji i ich wznawianie oparte na wykorzystaniu list ticketów używanych jako nonce – oznacza to brak przesyłania ustalonego identyfikatora sesji i uniemożliwienie obserwatorom grupowania ramek. Jest to inne wykorzystanie ticketów niż to opisano w RFC 5077.
- Uproszczone procedury handshake, które m.in. nie przeprowadzają negocjacji algorytmów szyfrowania.
- Dwa poziomy zabezpieczeń połączenia:
  - **standardowy** - dostępny dla wszystkich klientów. Zabezpieczenie połączenia następuje poprzez wykorzystanie wspólnego sekretu otrzymanego z procedury handshake wykorzystującej ECDHE (lub ECDH, gdy klient posiada autoryzowany klucz publiczny serwera).
  - **zalogowany** – poziom dostępny dla posiadaczy konta na danym serwerze. Logowanie w PrivMX to procedura, w której klient po wstępnym nawiązaniu „połączenia standardowego" wymusza przeprowadzenie dodatkowej procedury handshake opartej na protokole SRP. Moduł PrivMX TLS ma tutaj dostęp do danych zapisanych na serwerze podczas tworzenia konta użytkownika.
- Po ustanowieniu połączenia każda ze stron może w dowolnym momencie wymuszać zmianę kluczy szyfrowania połączenia.

Serwer PrivMX udostępnia klientom różne zestawy metod API w zależności od wykorzystywanego poziomu połączenia PrivMX TLS. Temat ten zaznaczany jest często w kolejnych rozdziałach, przy opisach metod API.

2.4 PrivMX Proxy

Serwer PrivMX udostępnia metodę `API.proxy`, która pozwala programom klienckim wykorzystać „ich własny" moduł serwerowy jako pośrednika w połączeniach z innymi serwerami PrivMX. Działanie tej usługi może być konfigurowane w zależności od potrzeb. W szczególności, od sposobu implementacji aplikacji klienckiej zależy czy i jak korzysta ona z tej możliwości.

Nawiązywanie połączeń PrivMX TLS z „obcym" serwerem poprzez usługę PrivMX Proxy nie zaburza bezpieczeństwa połączeń. Klucze szyfrowania ustalane są (handshake) z docelowym serwerem w taki sposób, że serwer pośredniczący nie ma do nich dostępu.

3. Serwer PrivMX

3.1 Główne funkcje API serwera

Moduł serwerowy PrivMX udostępnia programom klienckim metody API pozwalające na realizację następujących funkcji:

- **przechowywanie danych** - funkcje zapisu i odczytu bloków danych i deskryptorów;
- **obsługa skrzynek komunikacyjnych** oraz funkcje przyjmowania i odczytywania wiadomości;
- **magazyn i weryfikacja kluczy publicznych** - funkcje PrivMX PKI opisane są osobno w ostatnim rozdziale niniejszego dokumentu;
- **tworzenie kont, logowanie, proxy, profile użytkowników**;
- **administracja** – opis funkcji API związanych z zarządzaniem serwerem PrivMX jest w tym dokumencie pominięty.

W tym rozdziale skupiamy się na dwóch pierwszych zagadnieniach.

3.2 Klucze ECC - identyfikatory i prawa dostępu

Architektura PrivMX zakłada, iż identyfikatory dla nowo tworzonych obiektów nadawane są w większości przypadków przez program kliencki.

- Identyfikatorami najważniejszych obiektów są (zserializowane) klucze publiczne ECC lub liczone z nich 160 bitowe adresy bitcoin. Powstają one na bazie kluczy prywatnych ECC losowanych przez program kliencki;
- serwer NIE umożliwia programom klienckim enumeracji obiektów, które przechowuje.
- dostęp do „zwykłych", „nie chronionych" operacji na danych wymaga od programu klienckiego posiadania ich identyfikatora, czyli klucza publicznego ECC. Natomiast dostęp do określonych, „chronionych" operacji tj. modyfikacja obiektu lub jego usunięcie (a nawet czasem odczyt!) wymaga posiadania dodatkowo odpowiedniego klucza prywatnego ECC;

Tworzenie obiektów na serwerze PrivMX następuje zawsze tylko i wyłącznie na życzenie programu klienckiego. Ze względu na brak możliwości enumeracji obiektów, para kluczy ECC (privkey,pubkey) losowana przez moduł kliencki staje się jedyną drogą dostępu do obiektu. Przechowywanie tej pary to odpowiedzialność programu klienckiego – architektura PrivMX nie specyfikuje konkretnego sposobu przechowywania tych kluczy, jednak standardowa biblioteka kliencka zawiera pewne gotowe rozwiązania (p. dalsze rozdziały).

3.3 Bloki danych

Serwer PrivMX przyjmuje i serwuje dane podobnie jak sterowniki dysków w systemach plików - w postaci bloków o maksymalnej wielkości 128KB (wielkość ta może być konfigurowana).

Bloki to „porcje surowych danych" z przydzielonymi identyfikatorami - BIDs. Bloki tworzone są przez moduł kliencki, który przesyła je na serwer, gdzie są przechowywane bez modyfikacji zawartości.

- serwer nie szyfruje danych - odpowiedzialny za to jest program kliencki.
- Program kliencki ustala BID bloku tak, że jest on równy SHA256 zawartości przesyłanego bloku.

Ze względu na konstrukcję bloków (BID bloku = hash jego zawartości):

- moduły kliencki i serwerowy mogą korzystać z naturalnej kontroli spójności przesyłanych danych - identyfikator bloku to zarazem suma kontrolna danych;
- nie można stworzyć na serwerze dwóch bloków, dla których przesłane zostały te same dane;
- bloki są niemodyfikowalne (immutable) - „zmiana danych bloku" może być zrealizowana jedynie poprzez stworzenie i przesłanie nowego bloku – nowe dane oznaczają nowy BID.

Bloki nie istnieją samodzielnie – mogą być one tworzone i czytane przez moduł kliencki zawsze w powiązaniu z operacją na obiekcie, który z tych bloków korzysta – p. deskryptory i wiadomości poniżej. Każdy blok może być użyty przez kilka takich obiektów. Architektura PrivMX zakłada, iż bloki, które nie są przypisane do żadnego obiektu powinny być automatycznie usuwane przez serwer.

3.4 Deskryptory

Deskryptory umożliwiają zapisanie na serwerze większych porcji danych niż 128KB – deskryptory „grupują bloki" oraz stanowią punkt dostępu do nich dla programu klienckiego. Są również obiektami realizującymi podstawową politykę praw dostępu do danych (zgodną z wspomnianym wcześniej

sposobem wykorzystania publicznych i prywatnych kluczy ECC). Poniżej opisujemy to dokładniej.

Tworzenie deskryptorów `API.descriptorCreate*` i ich modyfikacja `API.descriptorUpdate*` wymaga zazwyczaj przesyłania jednego lub wielu bloków danych poprzez jedno- lub wielorazowe wykorzystanie odpowiedniej metody API. W tym celu wykorzystany jest mechanizm „sesji transferowych" - tworzony jest tymczasowy unikalny identyfikator transferu danych wykorzystywany w kolejnych wywołaniach `API.blockCreate`:

1. Program kliencki (zalogowany użytkownik) woła `API.descriptorCreateInit` i uzyskuje TransferID.
2. Następnie wysyła bloki używając `API.blockCreate` – podając TransferID oraz BID i dane każdego bloku. Jeśli program kliencki chce użyć bloków, które na serwerze już istnieją, to używa innej funkcji – `API.blockUseExisting`.
3. Losuje nowy klucz prywatny dpriv i generuje dla niego klucz publiczny dpub.
4. Na koniec tworzy na serwerze nowy deskryptor (`API.descriptorCreateFinish`). W tym celu przesyła mu:
    - DID (160bitów) - adres bitcoinowy (sieć „main") dla dpub;
    - TransferID, Blocks – wykorzystany TransferID oraz lista BIDs bloków przesłanych w trakcie sesji;
    - Extra - miejsce na dodatkowe, dowolne dane deskryptora, do wykorzystania przez aplikację klienc-ką.
    - dpub – klucz publiczny deskryptora;
    - podpis powyższych danych kluczem dpriv – żeby serwer wiedział, że klient ma odpowiedni klucz prywatny, tzn. że jest właścicielem deskryptora.

Deskryptory oraz bloki, w połączeniu z szyfrowaniem danych po stronie klienckiej (szyfrowanie danych bloków oraz pól Extra) umożliwiają stworzenie mechanizmów przechowywania danych, które ukrywają przed serwerem treść plików oraz strukturę katalogów – p. rozdział dot. Modułu klienckiego.

Odczytywanie deskryptora możliwe jest dla każdego programu klienckiego, który posiada odpowiedni DID. Wówczas wywołaniem `API.descriptorGet` klient uzyskać może pełen zestaw powyższych informacji (Blocks, Extra, dpub itd.) i kolejnymi wywołaniami `API.blockGet` pobrać może wszystkie bloki danych. Wywołanie `API.blockGet` wymaga podania BID oraz DID – nie wystarczy znajomość samego identyfikatora bloku, aby uzyskać dostęp do jego danych.

Modyfikacja deskryptora (`API.descriptorUpdate`) jest operacją podobną do tworzenia deskryptora - wymaga, aby żądanie było podpisane kluczem dpriv oraz wymaga przesłania nowych bloków (lub użycia istniejących). Żądanie usunięcia deskryptora (`API.descriptorDelete`) także wymaga podpisu kluczem prywatnym. Usunięcie deskryptora nie powoduje automatycznie usunięcia bloków – powinno się to dziać tylko wtedy, gdy nie są one używane przez inne deskryptory lub wiadomości.

3.5 Skrzynki pocztowe

Mechanizm działania skrzynek i przesyłania wiadomości pomiędzy programami klienckimi PrivMX (użytkownikami) można ogólnie opisać następująco:

1. **aby wysłać wiadomość**, program kliencki nadawcy musi ją podpisać kluczem nadawcy oraz nawiązać standardowe połączenie PrivMX TLS z serwerem adresata (może zrobić to korzystając z proxy). Uruchamia na tym serwerze metodę `API.messagePut`, aby wrzucić wiadomość do określonej skrzynki pocztowej. Skrzynki identyfikowane są poprzez klucze publiczne.
2. **Adresat odbiera wiadomość** ze swojego serwera, gdy program kliencki sprawdza stan swoich skrzynek (`API.sinkGetMessages`, `API.messageGet`). Odpowiednie żądania podpisać musi kluczami prywatnymi skrzynek.

Sposób przyjmowania wiadomości przez skrzynkę pocztową, określaną w terminologii PrivMX API jako „sink", ustala się w momencie jej tworzenia. Wówczas program kliencki, po wylosowaniu nowej pary kluczy (spriv, spub), do metody `API.sinkCreate` musi przekazać:

- SID, czyli zserializowany klucz publiczny spub.
- WriteMode - wartość „private", „public" lub „anonymous" określająca kto może wysyłać wiadomości do tej skrzynki:
    - **private** – tylko właściciel klucza prywatnego skrzynki może zostawiać wiadomości;
    - **public** – wiadomości zostawiać może dowolny użytkownik dowolnego serwera PrivMX. Podpis wiadomości jest wówczas weryfikowany poprzez wykorzystanie PrivMX PKI.
    - **anonymous** – skrzynka przyjmuje wiadomości podpisane dowolnym kluczem, np. takim, który jest wylosowany przed wysłaniem.
- Extra - pole do dowolnego wykorzystania przez program kliencki.
- Podpis powyższych danych wykonany kluczem spriv.

Pola WriteMode i Extra mogą być później przez program kliencki odczytywane (`API.sinkGetInfo`) i modyfikowane (`API.sinkUpdate`), a skrzynka jako całość może być usunięta poprzez wywołanie `API.sinkDelete`. Wszystkie wymienione tu operacje domyślnie dostępne są wyłącznie w połączeniu „zalogowanym" oraz muszą być podpisane odpowiednim kluczem prywatnym.

3.6 Wiadomości

Wysyłanie wiadomości (tzn. tworzenie i umieszczanie jej w określonej skrzynce docelowego serwera metodą `API.messsagePut`) podobne jest do tworzenia deskryptora – wiąże się z utworzeniem „sesji transferowej" i przesłaniem bloków danych. Jedynie wiadomości o wielkości do 1 MB można przesłać bez użycia bloków.

Zakładając, że program kliencki dysponuje parą kluczy nadawcy (cpriv,cpub):

1. przesyłanie wiadomości rozpoczyna wywołanie `API.messagePutInit` z parametrami:
   - SID skrzynki docelowej.
   - SenderAddress - adres nadawcy w formacie "user#server.net". To pole nie jest używane w przypadku skrzynek anonymous.
   - SenderPubKey – należący do nadawcy klucz publiczny cpub..
   - ExtraAuth – miejsce na opcjonalne dodatkowe dane do wykorzystania przez serwer do weryfikacji nadawcy. To pole przydatne jest zwłaszcza wtedy, gdy skrzynka działa w trybie anonymous – np. przyjmuje wiadomości z formularzy internetowych stosujących captcha.
   - Podpis całości żądania kluczem cpriv.
2. Gdy klient nie zostanie odrzucony (p. poniżej), to otrzymuje nowy TransferID, który może wykorzystać, aby wysłać bloki danych wiadomości. Robi to poprzez odpowiednie, następujące tutaj wywołania `API.blockCreate`, tj. w przypadku deskryptorów.
3. Przesyłanie wiadomości kończy wywołanie `API.messagePutFinish` z parametrami:
   - Extra – pole (max 1MB) do dowolnego wykorzystania przez klienta.
   - TransferID, Blocks – potwierdzenie sesji transferowej - wykorzystany TransferID i lista BIDs wysłanych bloków
   - Tags – lista dowolnych stringów ustawianych przez nadawcę. Serwer przechowuje je razem z wiadomoscią.
   - Podpis całości żądania kluczem cpriv.

Odrzucenie wiadomości nastąpić może z powodu złej weryfikacji adresu lub klucza nadawcy (PrivMX PKI) albo z powodu niepowodzenia opcjonalnej weryfikacji ExtraAuth. Serwer domyślnie nie sprawdza danych przesyłanych w polu ExtraAuth, ale mogą robić to rozszerzenia serwera – w zależności od konkretnej aplikacji.

Skrzynka numeruje przyjęte wiadomości i udostępnia numer ostatniej wiadomości (lastNumber) poprzez wspomniane już `API.sinkGetInfo`. Odczytywanie listy wiadomości znajdujących się w skrzynce dostępne jest dla posiadacza klucza prywatnego skrzynki poprzez odpowiednio podpisane wywołanie `API.sinkGetMessages`. Metoda ta pozwala klientowi uzyskać identyfikatory wiadomości (MIDs) z podanego przedziału numeracji oraz takich, które mają ustawione określone tagi.

Program kliencki może odczytać wiadomość poprzez wywołanie `API.messageGet` używając odpowiedniego MID oraz SID. Uzyskuje wówczas dostęp do pól wiadomości SenderAddress, SenderPubKey, ExtraAnon, Blocks, Extra, Tags. Żądanie odczytania wiadomości musi być podpisane kluczem prywatnym skrzynki. Metoda `API.messageDelete` wymaga podania podobnych danych i podpisu.

3.7 Publiczne i prywatne dane użytkowników

Oprócz obiektów opisanych w poprzednich rozdziałach, serwer przechowuje również dane powiązane z konkretnymi użytkownikami. Ich zakres zależny jest od konkretnych zastosowań.

W domyślnej konfiguracji serwer PrivMX dla każdego użytkownika przechowuje następujące dane prywatne generowane przez program kliencki:

- dane związane z logowaniem – parametry hashowania (sól, ilość rund, nazwa algorytmu hashowania) wykorzystywane po stronie klienckiej oraz weryfikator SRP. Serwer nie przechowuje hasła użytkownika.
- zaszyfrowany po stronie klienckiej niewielki rekord danych PrivData – są to dane nieczytelne dla serwera.

Korzystając z mechanizmu PrivMX PKI, serwer domyślnie udostępnia publicznie, tzn. dla dowolnych programów klienckich, następujące dane związane z użytkownikiem:

- SID domyślnej skrzynki pocztowej – powiązanej z adresem użytkownika;
- IdentityKeyPub, czyli klucz publiczny użytkownika;
- Dane profilowe użytkownika – opcjonalne dane takie jak imię, opis, avatar, itp.

Znaczenie powyższych danych oraz sposób dostępu do nich opisane są w dalszym ciągu dokumentu.

3.8 Uwagi końcowe

W tym podrozdziale zgromadzone zostały różnorodne uwagi dot. konfiguracji, rozszerzeń i implementacji serwera PrivMX.

Przechowywanie bloków, deskryptorów, skrzynek i wiadomości zrealizować można najlepiej poprzez wykorzystanie prostych i szybkich baz typu key-value, baz dokumentowych.

Usuwanie bloków - wymóg automatycznego usuwania bloków, które nie są przypisane do żadnego obiektu (deskryptora, wiadomości) zrealizować można np. poprzez cykliczne uruchamianie garbage-collectora. Możliwe jest ponadto rozszerzenie API o metodę `API.blockDelete` i wówczas program kliencki mógłby przechowywać BIDs bloków w (zaszyfrowanym) polu Extra deskryptora.

Istnieje sporo możliwości konfiguracji serwera PrivMX w zakresie dopuszczalności połączeń – od zablokowania dostępu do konkretnych metod API w ramach połączenia standardowego, poprzez ograniczenie połączeń pomiędzy serwerami, aż po wyznaczenie konkretnych adresów IP i użytkowników, które mogą kontaktować się z serwerem.

Domyślna implementacja serwera zakłada, że wszystkie funkcje API związane ze skrzynkami pocztowymi dostępne są jedynie w połączeniu zalogowanym. Oznacza to możliwość rozszerzenia API o metody skrzynek pocztowych „automatycznie" wykorzystujące klucze zalogowanego użytkownika. Wówczas nie byłoby konieczności ciągłego podpisywania żądań i przekazywania klucza publicznego.

Skrzynki pocztowe numerują trafiające do nich wiadomości i udostępniają ich liczbę - lastNumber. Istnieje możliwość implementacji dodatkowych „liczników", które mogą zoptymalizować proces sprawdzania skrzynek, odczytywania i modyfikowania wiadomości.

Serwer może zaznaczać sobie i przechowywać informację o tym kto utworzył (i jest pierwszym właścicielem) obiektów takich jak bloki, deskryptory, skrzynki.

4. Standardowa biblioteka kliencka PrivMX

4.1 Informacje wstępne

Moduł kliencki to centralna część systemów opartych o architekturę PrivMX. Serwer oferuje ograniczoną funkcjonalność i domyślnie to aplikacja kliencka realizuje całość logiki biznesowej systemu dbając jednocześnie o odpowiednie szyfrowanie danych.

Standardowa biblioteka kliencka PrivMX dostarcza podstawowych elementów do budowania logiki systemu w oparciu o:

- mechanizmy bezpiecznego tworzenia kont i logowania;
- obiekty „wysokopoziomowe" takie, jak katalogi, pliki, skrzynki nadawcze, odbiorcze itd.;
- udostępnianie danych;
- szyfrowanie danych przed wysłaniem na serwer, połączone z odpowiednim generowaniem i przechowywaniem kluczy.

W tym rozdziale opisujemy szkicowo powyższe funkcje.

4.2 Rozszerzone klucze ECC

Standardowy moduł kliencki PrivMX wykorzystuje tzw. rozszerzone klucze ECC (p. specyfikacja BIP 32) - są to zwykłe klucze privkey 256bit i pubkey 512bit (spakowane do 257bit) rozbudowane o dodatkową daną, tzw chaincode o długości 256 bitów - „dodatkowe źródło entropii". Klucze rozszerzone zapisujemy w tym dokumencie z plusem - jako privkey+chaincode oraz pubkey+chaincode.

Chaincodes wykorzystywane są przez klienta PrivMX w procesie derywowania kluczy pochodnych z kluczy istniejących (funkcja CKD z BIP 32) oraz jako klucze do szyfrowania danych użytkownika algorytmem symetrycznym (AES).

4.3 Tworzenie kont

Protokół PrivMX TLS pozwala aplikacjom klienckim na logowanie, tzn. wymuszenie przeprowadzenia dodatkowej procedury handshake opartej na SRP. Żeby było to możliwe, serwer musi być w posiadaniu weryfikatora SRP. Zapewnienie tego jest jednym z celów procedury tworzenia konta. Innym celem tej procedury jest ustalenie i zapamiętanie klucza prywatnego i publicznego dla nowego użytkownika.

Nowe konta użytkowników na serwerze PrivMX domyślnie tworzone mogą być przez programy klienckie jedynie na bazie „zaproszeń" - 32-bajtowych, losowych tokenów. Do generowania tokenów służy metoda `API.generateNewUserToken` dostępna w połączeniu zalogowanym dla wyróżnionych użytkowników („administratorów"). Pierwszy token dla pierwszego użytkownika generowany jest zazwyczaj podczas procedury instalacji serwera PrivMX.

Dowolny, połączony standardowo program kliencki może rozpocząć procedurę tworzenia konta, o ile posiada ważny token oraz poprosił już użytkownika o podanie jego nazwy i hasła.

1. Program kliencki losuje Sól (16 bajtów) oraz IlośćRund (liczba z przedziału 4000-5000);
2. wyznacza MixedPassword = H(hasło użytkownika, Sól, IlośćRund), gdzie H to ustalony przez program kliencki algorytm hashowania, np. PBKDF2-SHA512;
3. oblicza srpVerifier wykorzystując wartość hash = SHA512(MixedPassword) % 16 bajtów;
4. losuje MasterKey = rozszerzony prywatny klucz ECC;
5. ustala privData = AES256Encrypt(MasterKey) z hasłem równym SHA256(MixedPassword);
6. oblicza identityKey = CKD(MasterKey, m/0'), a następnie identityKeyPub = rozszerzony klucz publiczny liczony dla identityKey;
7. wywołuje metodę `API.register`, do której przekazuje:
   - token otrzymany od administratora;
   - nazwę nowego użytkownika;
   - srpVerifier – będzie wykorzystywany przez serwer podczas późniejszego logowania;
   - Sól, IlośćRund i nazwę algorytmu hashowania – serwer dostarczy te dane każdemu klientowi, który będzie chciał się zalogować na to konto;
   - privData, które klient będzie mógł bezpiecznie pobrać po zalogowaniu (p.poniżej);
   - identityKeyPub – klucz publiczny nowego użytkownika, który będzie opublikowany przez serwer w ramach usługi PrivMX PKI;
   - signature – podpis zestawu wszystkich powyższych danych wykonany kluczem prywatnym identityKey.

Serwer zapamiętuje powyższe dane i korzysta z nich przy późniejszych próbach logowania na nowo utworzone konto.

4.4 Logowanie i inicjalizacja klienta

Aplikacja kliencka korzystająca z modułu PrivMX swoje działanie rozpoczyna najczęściej od uzyskania standardowego połączenia z serwerem PrivMX. W tym celu łączy się z ustalonym endpointem API lub wykorzystuje najpierw procedurę Service Discovery dla podanego hostname lub adresu user#hostname.

Jeśli aplikacja chce uzyskać dostęp do danych określonego użytkownika lub chce wysyłać w jego imieniu wiadomości – musi przeprowadzić procedurę logowania. Zakładając, że klient nawiązał już standardowe połączenie PrivMX TLS i posiada podane przez użytkownika jego identyfikator i hasło:

1. klient pobiera z serwera Sól i IlośćRund zapisane dla danej nazwy użytkownika (`API.getLoginParams`);
2. korzystając z tych danych oraz hasła użytkownika oblicza MixedPassword oraz srpVerifier – tak samo jak podczas zakładania konta;
3. wymusza odpowiednią funkcją PrivMX TLS przeprowadzenie hadshake SRP, w wyniku czego zmieniają się klucze szyfrowania połączenia oraz uzyskuje autoryzację jako użytkownik posiadający konto – otrzymuje dostęp do metod API dla zalogowanych użytkowników;
4. pobiera z serwera privData użytkownika (`API.getPrivData`);
5. oblicza MasterKey = AES256Decrypt(privData) z hasłem równym SHA256(MixedPassword);
6. oblicza identityKey oraz identityKeyPub – na bazie MasterKey, w taki sam sposób jak podczas zakładania konta.

Dwa klucze uzyskane w wyniku przeprowadzenia powyższej procedury stanowią podstawę do dalszego działania standardowego modułu klienckiego:

- **rozszerzony klucz prywatny MasterKey** stanowi dla programu klienckiego „główny uchwyt” do danych użytkownika zapisanych na serwerze – generowane są z niego:
  - rozszerzony klucz prywatny **HomeDir** = CKD(MasterKey,m/1’) wyznaczający i dający pełen dostęp do „katalogu domowego” użytkownika;
  - rozszerzony klucz prywatny **SinkList** = CKD(MasterKey,m/2) wyznaczający i dający pełen dostęp do pliku przechowującego klucze prywatne do skrzynek pocztowych użytkownika.
- **rozszerzony klucz prywatny IdentityKey** = CKD(MasterKey,m/0’) to klucz prywatny użytkownika, który wykorzystywany jest do odszyfrowywania wiadomości skierowanych do użytkownika, do potwierdzania jego tożsamości (podpisy) itp.

Wszystkie wyżej wymienione klucze generowane są przez program kliencki i nie są przechowywane po stronie serwera.

Po przeprowadzeniu pierwszego logowania standardowy moduł kliencki nie może odczytać katalogu HomeDir oraz pliku SinkList, ponieważ one jeszcze nie istnieją. Musi je utworzyć i może wypełnić je danymi początkowymi zależnie od aplikacji. Domyślne działanie jest takie, że tworzony jest wówczas pusty katalog domowy oraz pierwsza skrzynka pocztowa, której klucz prywatny umieszczany jest w pliku SinkList. Klucz publiczny skrzynki może być ustalony jako domyślny publiczny SID użytkownika powiązany z adresem nazwa#hostname.

4.5 Pliki i katalogi

Serwer PrivMX nie oferuje funkcji API związanych z obiektami takimi jak pliki, czy katalogi. Jeżeli aplikacja kliencka chce wykorzystywać takie obiekty i przechowywać je na serwerze PrivMX, to musi je symulować przy pomocy dostępnych metod API. Standardowy moduł kliencki robi to przyjmując następujące założenia:

- **plik** to deskryptor z powiązanymi blokami. Dane pliku przechowywane są w blokach, a metadane pliku w polu Extra deskryptora.
- **Katalog** to specjalny plik (w formacie json) zawierający listę nazw i kluczy plików oraz podkatalogów, które w danym katalogu mają się znajdować. Pole Extra deskryptora zawiera metadane katalogu.

Klucz HomeDirKey uzyskiwany po logowaniu to rozszerzony klucz prywatny katalogu głównego, którego bloki zawierają listę wszystkich plików i katalogów “głównego poziomu” - ich identyfikatory, klucze publiczne i prywatne. W ten sposób klient PrivMX uzyskuje strukturę drzewiastą zaszyfrowanych plików i katalogów.

Struktura ta oraz dane plików są ukryte przed serwerem, ponieważ standardowy moduł kliencki PrivMX szyfruje dane, które umieszcza w blokach oraz w polach Extra deskryptorów. Stosuje w tym celu szyfrowanie symetryczne - algorytm AES256 - jako klucze stosując chaincodes z rozszerzonych kluczy ECC.

Przykładowo: aby zapisać plik w katalogu k, program kliencki musi posiadać rozszerzony klucz prywatny tego katalogu (kpriv+kchaincode), żeby udowodnić serwerowi, że ma prawa do jego zapisu/modyfikacji. Wykonuje wówczas procedurę tworzenia nowego deskryptora z odpowiednio zaszyfrowanymi danymi oraz dopisania go do katalogu k:

1. Losuje 256-bitowy klucz KS do szyfrowania danych pliku.
2. Dzieli plik na bloki, szyfruje każdy z nich kluczem KS i nadaje im BID (hash szyfrogramu).
3. Losuje nowy rozszerzony klucz prywatny (dpriv+dchaincode) i generuje z niego rozszerzony klucz publiczny (dpub+dchaincode) – identyfikator nowego pliku.
4. Przygotowuje strukturę Metadata zawierającą:
   - type = „file”

- data = { nazwapliku, mimetype, rozmiar, data utworzenia i modyfikacji pliku }
- blockskey = KS

5. Tworzy na serwerze nowy deskryptor (p. poprzednie rozdziały) przesyłając przygotowane bloki, klucz dpub, odpowiedni DID oraz:
   - Extra = AES256Encrypt(data=Metadata, key=dchaincode)
   - Podpis całości żądania wykonany kluczem dpriv
6. Korzystając z klucza prywatnego kpriv modyfikuje katalog/deskryptor k tak, że dopisuje do jego danych (json) linię-czwórkę zawierającą:
   - name = nazwapliku
   - type = ”file”
   - pub = (dpub+dchaincode)
   - encpriv = AES256Encrypt(data=dpriv,key=kpriv)

Możliwość odczytania danych związanych z deskryptorem pliku/katalogu zależna jest od kluczy, które posiada program kliencki:

- **Posiadanie samego DID** nie umożliwia w zasadzie dostępu do danych. Pozwala sprawdzić, czy dany deskryptor istnieje i dzięki dostępowi do pola Blocks pozwala pobrać bloki zaszyfrowanych danych.
- **Posiadanie rozszerzonego klucza publicznego (dpub+dchaincode)** pozwala na powyższe oraz dodatkowo na odszyfrowanie pola Extra, czyli uzyskanie metadanych pliku oraz klucza blockskey. Dzięki temu ostatniemu klient może odszyfrować bloki danych – otrzyma wówczas odszyfrowane dane pliku. W przypadku katalogu otrzyma listę plików i podkatalogów, które się w nim znajdują i tym samym możliwość czytania katalogu „wgłąb”.
- **Posiadanie rozszerzonego klucza prywatnego (dpriv+dchaincode)** pozwala odczytać wszystkie dane jw. oraz dodatkowo umożliwia programowi klienckiemu modyfikację deskryptora i jego usunięcie. W przypadku gdy jest to katalog – możliwe jest rozszyfrowanie kluczy prywatnych plików i podkatalogów oraz w konsekwencji ich modyfikacja i usuwanie.

4.6 Wysyłanie i odbieranie wiadomości

Po przeprowadzeniu procedury logowania standardowa biblioteka kliencka PrivMX uzyskuje klucz SinkList do pliku zawierającego listę skrzynek pocztowych użytkownika. Dla każdej skrzynki znaleźć tu można jej nazwę, opis oraz rozszerzony klucz prywatny. Plik ten jest odpowiednio aktualizowany po utworzeniu nowej lub usunięciu niepotrzebnej skrzynki pocztowej.

Program kliencki musi znać identyfikator skrzynki (SID) adresata, aby wysłać do niego wiadomość. Aplikacja kliencka, w zależności od wymagań może dbać na różne sposoby o to, żeby użytkownicy znali SID swoich skrzynek pocztowych.

Domyślnie zakłada się zdecentralizowaną konfigurację systemu opartego o architekturę PrivMX i dlatego standardowy moduł kliencki po pierwszym logowaniu tworzy skrzynkę pocztową dla użytkownika, dopisuje ją do SinkList i jej identyfikator umieszcza w publicznym rekordzie użytkownika w usłudze PrivMX PKI. Przy pomocy tej usługi nadawcy wiadomości mogą pobrać (i zweryfikować!) domyślny SID użytkownika. Więcej na ten temat przeczytać można w ostatnim rozdziale dokumentu.

Standardowy moduł kliencki wykorzystuje `API.messagePut` do pozostawienia swej wiadomości w odpowiedniej skrzynce na docelowym serwerze. Wykorzystuje jednak w specyficzny sposób bloki wiadomości i pole Extra:

- Bloki danych wykorzystywane są głównie do przesyłania plików – np. załączników do wiadomości. Dla każdego załącznika losowany jest nowy klucz, którym szyfrowane są bloki tego załącznika (AES256Encrypt).
- Główna struktura danych wiadomości umieszczana jest w polu Extra i szyfrowana jest wspólnym sekretem generowanym zgodnie ze schematem ECIES dla klucza prywatnego IdentityKey nadawcy i klucza publicznego SID skrzynki. Standardowa struktura wiadomości zawiera, między innymi:
  - tytuł i treść wiadomości – pola tekstowe formatowane dowolnie przez klienta;
  - senderName - nazwa nadawcy (np. „John Smith”)
  - attachments - lista załączników; dla każdego określone są:
    - nazwa i mimetype;
    - blocks - BIDs bloków tego załącznika;
    - key - klucz szyfrujący te bloki.

Zastosowanie w schemacie ECIES klucza publicznego skrzynki pocztowej (SID) powoduje, że treść przesyłanych danych będzie mogła być pobrana, rozszyfrowana i odczytana tylko poprzez posiadacza klucza prywatnego docelowej skrzynki pocztowej. Jeśli jest to np. skrzynka domyślna użytkownika, który nie dzielił się z nikim swoimi kluczami (co jest najczęstszym przypadkiem), to wiadomość odczyta tylko ten użytkownik.

4.7 Udostępnianie danych

Standardowy sposób udostępniania danych w ramach architektury PrivMX polega na przekazywaniu odpowiednich kluczy do określonych obiektów przechowywanych przez serwer. Chcąc np. udostępnić innemu użytkownikowi plik tylko do odczytu - należy przekazać mu klucz publiczny do niego.

Z klucza prywatnego ECC oblicza się odpowiedni klucz publiczny, zatem na przykład przekazując klucz prywatny do deskryptora, dajemy odbiorcy możliwość jego modyfikacji, ale też i odczytu. W przypadku skrzynek pocztowych otrzymanie prawa do odczytu (klucz prywatny) daje zarazem prawo do umieszczania w niej nowych wiadomości.

Sposób dystrybucji kluczy zależy w pełni od aplikacji klienckiej, jej przeznaczenia, sposobu działania i rozwiązań interfejsu użytkownika. Klucze do wspólnych obiektów (najczęściej plików i katalogów) rozsyłane są najczęściej wiadomościami do domyślnych lub specjalnych skrzynek użytkowników. Takie podejście umożliwia również udostępnianie danych pomiędzy użytkownikami z różnych serwerów, aczkolwiek warunkiem jest tu odpowiednia konfiguracja serwerów dopuszczająca dostęp do danych również „obcym użytkownikom".

4.8 Uwagi nt. implementacji i modyfikacji klienta

Wszystkie klucze wykorzystywane przez system oparty na architekturze PrivMX są losowane przez moduł kliencki, dlatego jednym z najważniejszych elementów wpływającym bezpieczeństwo takiego systemu jest wysoka jakość generatora liczb losowych. Każda implementacja modułu klienckiego PrivMX powinna o to zadbać.

Procedury zakładania konta i logowania mogą być rozszerzone o wykorzystanie schematu 2-factor authentication.

Grupowanie żądań to optymalizacja protokołu PrivMX TLS, która usprawnia przeprowadzanie operacji na plikach i katalogach wymagających dokonania wielu zapytań API. Oczywiście optymalizacja taka wymaga również wsparcia po stronie serwera.

Derywowany domyślnie po zalogowaniu klucz SinkList do pliku zawierającego listę prywatnych skrzynek może być zastąpiony zwykłym plikiem o ustalonej nazwie w katalogu HomeDir.

Możliwe jest również przechowywanie kluczy prywatnych skrzynek pocztowych w nich samych – w ich polu Extra. Może to być przydatna optymalizacja, gdy API serwera rozszerzone jest o funkcje listujące wszystkie skrzynki danego użytkownika. Klucz prywatny skrzynki powinien być wówczas zaszyfrowany chainkodem klucza HomeDir (na przykład).

Domyślnie, poprzez procedurę derywacji kluczy CKD uzyskiwane są tylko „główne mountpointy" dla użytkownika, ale aplikacja kliencka może w ten sposób tworzyć na własne potrzeby dowolne, większe hierarchie kluczy prywatnych.

Programy klienckie mogą tworzyć wiadomości, które są możliwe do odszyfrowania tylko przez określonych użytkowników (a nie przez dowolnego właściciela klucza prywatnego skrzynki). W tym celu w treści wiadomości umieszczać mogą część zaszyfrowaną schematem ECIES z wykorzystaniem klucza prywatnego nadawcy i klucza publicznego odbiorcy (a nie skrzynki).

Parametry konfiguracji serwera, które są istotne dla programów klienckich (np. maksymalna wielkość bloku danych) dostępne są do odczytu metodą `API.getServerConfig`.

5. PrivMX PKI

5.1 Infrastruktura kluczy publicznych PrivMX

PrivMX PKI to elementy architektury PrivMX realizujące publikację i weryfikację kluczy publicznych użytkowników i serwerów. Mają one znaczenie przede wszystkim w przypadku systemów zdecentralizowanych i umożliwiają m.in. weryfikację nadawców wiadomości – sprawdzenie czy odebrana właśnie wiadomość zawiera poprawny podpis nadawcy.

PrivMX nie wykorzystuje w tym celu rozwiązań klasycznego PKI zakładającego wykorzystanie zewnętrznych urzędów certyfikacji. Nie wykorzystuje również rozwiązań PGP opartych o wzajemne podpisywanie kluczy publikowanych na zewnętrznych serwerach.

Infrastruktura kluczy publicznych PrivMX bazuje na ideach związanych ze specyfikacją CONIKS (https://coniks.org), dzięki czemu spełnione może być jedno z założeń bezpiecznej komunikacji PrivMX – likwidacja lub minimalizacja roli „zaufanej trzeciej strony" kontrolowanej przez firmy/organizacje zewnętrzne względem aplikacji.

W przypadku aplikacji zdecentralizowanych, w domyślnej konfiguracji każdy serwer PrivMX prowadzi własną bazę kluczy publicznych (keystores) swoich użytkowników i udostępnia ją poprzez API programom klienckim.

Bazy kluczy PrivMX to prywatne bazy publikujące hash (w sensie „skrót kryptograficzny" SHA256) całej swojej zawartości oraz historię zmian tego hasha przy pomocy struktury typu append-only (tj. blockchain). Dzięki temu bazy te mogą podlegać audytowi i monitorowaniu spójności przeprowadzanemu przez inne serwery PrivMX. Każdy pobierany klucz publiczny (keystore) dostarczany jest razem z dowodem poprawności, który powiązany jest z hashem całości bazy. Weryfikacja poprawności pobranego keystore składa się z dwóch kroków – sprawdzenie poprawności hasha bazy w oparciu o jego historię (przeprowadzane również wspólnie z innymi serwerami – audyty, web-of-trust) oraz sprawdzenie dostarczonego dowodu poprawności keystore.

Poniższe podrozdziały opisują nieco dokładniej cechy i działanie PrivMX PKI oraz domyślny sposób wykorzystania tych mechanizmów w aplikacji zdecentralizowanej.

5.2 Prywatna baza kluczy z publiczną historią zmian

„KeyStore" to nazwa, która w PrivMX oznacza „pęk kluczy". Architektura PrivMX zakłada obsługę dwóch typów keystores:

- **pgpkeystore** – keystore kompatybilny z PGP, zawierający pełną historię zmian, pozwalający na wykonywanie na jego danych operacji takich jak unieważnianie kluczy (revoke) oraz dodawanie podkluczy (i innych pakietów PGP tj public key, signature, userid). Zmian keystore tego typu dokonywać może posiadacz wyróżnionego klucza prywatnego.
- **simplekeystore** – keystore o prostej konstrukcji, zawierający listę kluczy publicznych. Taki keystore może być modyfikowany przy pomocy dowolnego, pasującego klucza prywatnego. Historia zmian keystore tego typu nie jest w nim pamiętana.

Keystores oprócz danych kluczy i informacji o swoim typie zawierać mogą również załączniki będące dowolnymi plikami.

Standardowe implementacje serwera i klienta PrivMX dostarczają odpowiednich funkcji do posługiwania się obydwoma typami keystores. Pozwalają one na ich tworzenie, czytanie i modyfikowanie przy użyciu kluczy prywatnych użytkowników. W tym dokumencie pomijamy opis tych funkcji.

Zbiór (baza) keystores przechowywanych przez serwer PrivMX ma strukturę binarnego drzewa Merkle. W uproszczeniu przedstawić można to tak, że liście drzewa to keystores, a węzły inne niż liście składają się z hashy SHA256 swoich potomków. Hash korzenia drzewa jest zatem hashem całości bazy - nazywany jest on często „rewizją drzewa" lub „rewizją bazy".

Każda zmiana w drzewie (tzn. dodanie lub modyfikacja keystore) powoduje zmiany na odpowiednich gałęziach drzewa oraz zmianę hasha całości bazy. Oprócz historii zmian drzewa, PrivMX PKI przechowuje również historię zmian hasha-rewizji całości bazy w postaci listy typu append-only – każdy element tej historii zawiera kolejny hash bazy oraz hash elementu poprzedniego w historii.

Ta lista, podobna do struktury blockchain, jest przez serwer PrivMX udostępniana publicznie (`API.pkiGetHistory`). Nie zawiera ona w sobie żadnych danych (kluczy i załączników), które są przechowywane przez PrivMX PKI, ale wystarcza do przeprowadzenia weryfikacji kluczy pobieranych z danego serwera oraz do przeprowadzania zewnętrznych audytów bazy.

5.3 Pobieranie i weryfikacja kluczy publicznych

Dostęp do keystores na poziomie API serwera odbywa się poprzez zastosowanie czytelnych identyfikatorów alfanumerycznych takich jak „server:simplito.com", „user:john#example.com". Identyfikatory takie przekształcane są przez serwer na zero-jedynkowe ścieżki w drzewie binarnym przy pomocy weryfikowalnej funkcji losowej (Verifiable Random Function). PrivMX PKI stosuje ścieżki-identyfikatory o ustalonej długości 256 bitów oraz przeprowadza ich kompresję, aby uniknąć tworzenia drzew o zbyt dużej głębokości.

Argumentami funkcji `API.pkiKeyStoreGet` są:

- name – identyfikator alfanumeryczny zawierający dwukropek (wyznacza on namespaces);
- includeAttachments – jeśli równe true (1), to funkcja oprócz danych kluczy zwraca również wszystkie załączniki wybranego keystore;
- revision – hash (rewizja) bazy, z której pobrany ma zostać keystore.

Funkcja zwraca dane keystore (jeśli odpowiedni keystore istnieje w bazie), dowód poprawności oraz załączniki, jeśli były wymagane. „Dowód poprawności" to generowany przez wspomnianą funkcję VRF ciąg zawierający odpowiednie fragmenty ścieżki binarnej oraz wszystkie hashe węzłów (w tym korzenia) potrzebne do tego, aby program kliencki mógł zweryfikować, że pobrany keystore należy do bazy wymaganej rewizji i że nie brakuje w nim żadnych danych. Odpowiednie funkcje realizujące tę weryfikację dostarczone są w standardowych implementacjach klienta PrivMX.

Pominięcie parametru revision podczas pobierania keystore powoduje, iż serwer wykorzysta rewizję bazy danych obowiązującą w momencie zapytania. Wówczas sprawdzenie dowodu poprawności wymaga dodatkowo pobrania historii bazy, aby odnaleźć w niej zawarty w dowodzie hash-rewizję bazy.

Pobieranie historii bazy oraz keystores możliwe jest po nawiązaniu „standardowego" połącznienia PrivMX TLS – jest domyślnie dostępne dla wszystkich programów klienckich. Serwer udostępnia również funkcje `API.pkiKeyStorePut`, `API.pkiKeyStoreModify`, `API.pkiKeyStoreDelete`, do których dostęp jest limitowany i konfigurowany w zależności od konkretnej aplikacji.

5.4 Audyty, konsensus, web-of-trust

Pomimo możliwości weryfikacji kluczy, która opisana jest powyżej, program kliencki musi „wierzyć" serwerowi, z którego je pobiera. W sytuacji aplikacji zdecentralizowanej i otwartej nie jest to kwestia oczywista, gdy wykorzystywanych jest wiele serwerów należących do różnych osób i organizacji. Aby uniknąć sytuacji takiej, że klientowi przedstawiana jest historia i klucze inne niż pozostałym klientom – można sprawdzić serwer przeprowadzając audyt.

Audyt polega na rozesłaniu do wybranych (innych) serwerów prośby o weryfikację historii bazy sprawdzanego serwera. Polega to na nawiązaniu połączenia i wywołaniu w wybranych serwerach `API.pkiConfirmRevision` podając jedynie:

- hostname – nazwa domeny serwera, który chcemy sprawdzić;
- revision – hash bazy, którego istnienie na docelowym serwerze chcemy potwierdzić.

Serwery proszone o audyt sprawdzają czy serwer hostname udostępnia im spójną historię zawierającą revision. Dokonują w tym celu odpowiednich wywołań `API.pkiGetHistory` oraz wykorzystują zapamiętaną wcześniej historię bazy serwera hostname. Każdy z proszonych o pomoc serwerów przekazuje ostatecznie informację, czy on również widzi revision na serwerze hostname, czy nie.

Po zebraniu tych informacji program kliencki może odpowiednio poinformować użytkownika o poziomie wiarygodności serwera, którego klucze pobiera i weryfikuje. Jeśli wszystkie odpowiedzi od audytorów są pozytywne lub wszystkie są negatywne, to możemy mówić o osiągnięciu porozumienia (konsensus) odnośnie wiarygodności sprawdzanego serwera. Wszystkie sytuacje pośrednie, gdy odpowiedzi są różne lub gdy niektóre serwery nie odpowiadają, wymagają skonfigurowania w aplikacji klienckiej odpowiedniej heurystyki wyznaczającej konsensus i/lub wyznaczenia operacji dostępnych dla użytkownika (oznaczenie serwera jako niezaufanego, ponowna próba weryfikacji, konsultacje z użytkownikami innych serwerów itp.).

Ilość i dobór serwerów-audytorów to dwie najważniejsze sprawy podczas zlecania audytów. Aplikacja, która je wykonuje może wykorzystywać ustalone, zaufane serwery PrivMX lub/oraz może pozwalać użytkownikom na budowanie własnych web-of-trust – sieci zaufanych serwerów. Może wybierać audytorów losowo lub według dowolnych własnych kryteriów. Architektura PrivMX nie determinuje sposobu dobierania serwerów, jest to zależne od konkretnej aplikacji.

5.5 Przykłady wykorzystania PrivMX PKI

Poniżej opisujemy działania, które domyślnie podejmuje serwer PrivMX konfigurowany do pracy w ramach aplikacji zdecentralizowanej (np. aplikacja PrivMX WebMail dostępna pod adresem https://privmx.com).

- Podczas instalacji serwera PrivMX ustalany jest klucz prywatny serwera, a odpowiedni klucz publiczny umieszczany jest w PrivMX PKI pod identyfikatorem „server:hostname".
- Każde wywołanie `API.register`, które rejestruje nowego użytkownika wstawia do bazy klucz publiczny użytkownika (IdentityKeyPub) z identyfikatorem „user:id#hostname". Do tego klucza dołączone są załączniki zawierające avatar użytkownika, SID jego publicznej skrzynki pocztowej oraz inne publiczne dane profilowe. Keystore ten wykorzystywany jest przez innych użytkowników m.in. do weryfikowania wiadomości podpisanych przez użytkownika id#hostname.
- Pierwszy użytkownik tworzący konto na nowym serwerze uzyskuje prawa administratora. W trakcie pierwszego logowania tego użytkownika do bazy dopisywany jest keystore „admin:hostname" (typu simplekeystore), którego rolą jest przechowywanie kluczy publicznych wszystkich administratorów serwera. Keystore ten udostępniany jest domyślnie wszystkim użytkownikom danego serwera, którzy mogą sprawdzić dzięki temu, czy np. dana wiadomość pochodzi od admina ich serwera.

Wspomniana aplikacja PrivMX WebMail pozwala dodatkowo administratorom serwerów na tworzenie własnych sieci web-of-trust. Wysyłają oni między sobą specjalne wiadomości-zaproszenia, których zaakceptowanie powoduje, że oba serwery dopisują się wzajemnie do swoich prywatnych list zaufanych serwerów. Listy te przechowywane są jako załączniki w keystore „admin:hostname" i są odczytywane przez każdego użytkownika podczas każdego logowania. W ten sposób administrator publikuje (proponuje) listy zaufanych serwerów swoim klientom. Dodatkowo, działanie funkcji `API.pkiRevisionConfirm` ograniczone jest na serwerze tak, że przeprowadza ona audyt tylko wtedy, gdy prosi ją o to serwer z listy zaufanych serwerów.