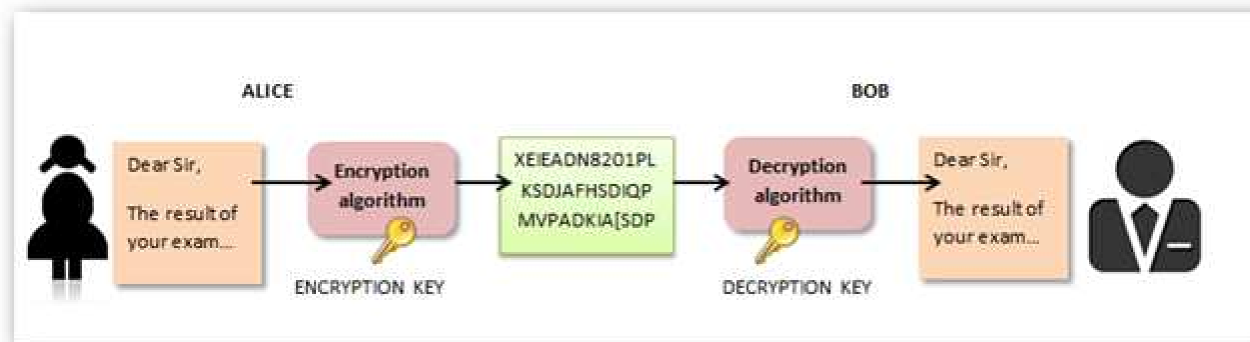
Szyfrowanie

Jedną z najskuteczniejszych technologii zapewnienia bezpieczeństwa współczesnych systemów komputerowych i sieciowych jest szyfrowanie. Jest stosowane w odniesieniu do pojedynczych dokumentów, jak i ciągłych transmisji sieciowych. Stosowane jest również do uwierzytelniania, zarówno integralności dokumentów jak i tożsamości osób.



Szyfrowanie było stosowane od starożytności i techniki szyfrowania mają długą historię rozwoju. Obecnie podstawową zasadą w szyfrowaniu jest, że nikt nie dąży do utajnienia algorytmu szyfrowania — te są ogólnie znane i dostępne. Tajny jest jedynie **klucz** szyfrowania, i im bardziej złożony (długi) jest ten klucz, tym trudniej złamać szyfr.

Można to porównać do technologii budowy zamków. Konstrukcje zamków są ogólnie znane, nie tworzy się zabezpieczeń przez wymyślne konstrukcje zamków. Zamiast tego, wybieramy jedną z dobrze znanych, solidnych konstrukcji zamków, oraz unikalny klucz.

Jawność algorytmów szyfrowania i długość klucza

Założenie jawności algorytmów szyfrowania może budzić wątpliwości.
Czy zaszkodziłoby zastosowanie nieznanego nikomu algorytmu szyfrowania?

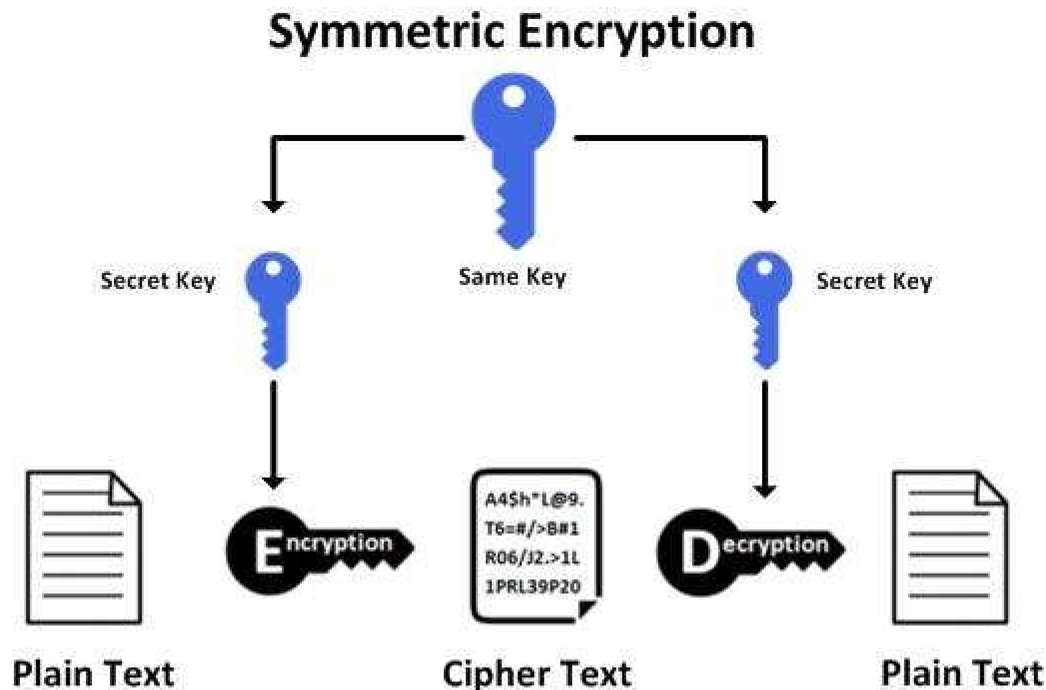
Paradoksalnie, TAK. Jest długa historia tajnych algorytmów szyfrowania, które zostały rozpracowane (ang. *reverse engineered*). Należą do nich: algorytm RC4, algorytmy szyfrowania protokołów transmisji komórkowej, algorytmy kodowania DVD i DIVX, i inne. Można bezpiecznie stwierdzić, że każdy tajny algorytm zostanie po jakimś czasie rozpracowany i ujawniony.

Z kolei jawne algorytmy szyfrowania są szczegółowo analizowane przez kryptografów na całym świecie, i wszelkie ich słabości zostają prędzej czy później odkrywane i ujawniane. Jawny, ogólnie znany algorytm szyfrowania, którego żadna słabość nie została ujawniona, najprawdopodobniej łatwo wykrywalnych słabości nie ma.

Czynnikiem, który zapewnia siłę nowoczesnych algorytmów szyfrowania są klucze. Uniwersalnym mechanizmem pozwalającym złamać każdy algorytm szyfrowania jest atak siłowy (*brute-force attack*), polegający na próbowaniu wszystkich kluczy po kolei. Istnieje 2^n możliwych kluczy o długości n bitów. Zatem liczba możliwych kluczy rośnie eksponencjalnie z długością klucza. Ale atak siłowy wymaga mocy obliczeniowej proporcjonalnej do liczby kluczy. Oznacza to, że dobry algorytm szyfrowania pozwala łatwo osiągnąć bezpieczeństwo szyfrowania. W praktyce, klucz o długości około 100 bitów zabezpiecza przeciwko wszystkim praktycznym atakom na miliony lat!

Szyfrowanie symetryczne

Najczęściej stosowane są algorytmy szyfrowania **symetrycznego**. Ich istotą jest identyczność kluczy (szyfrów) służących do szyfrowania i deszyfrowania.



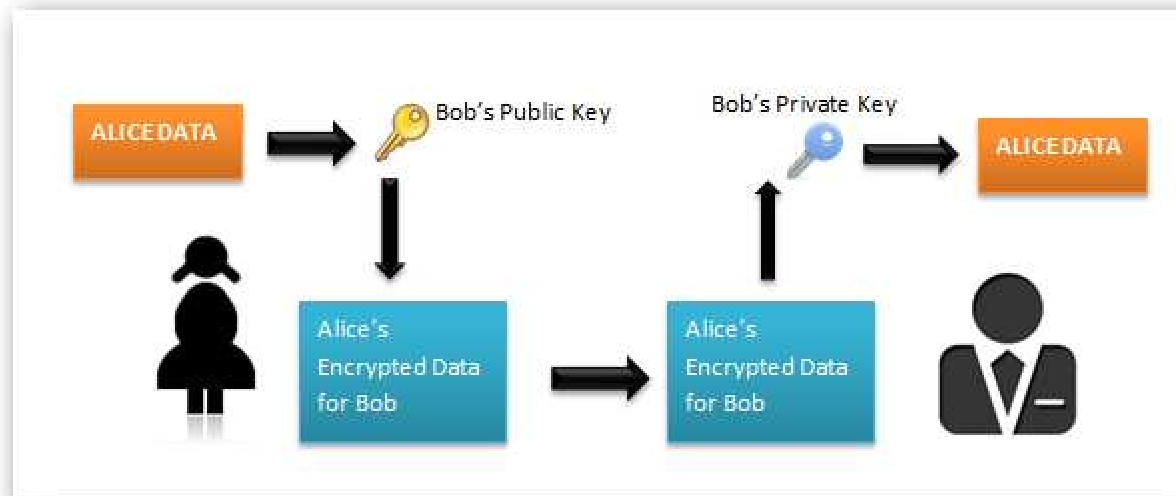
Bezpieczeństwo tego systemu opiera się na tajności kluczy, co stanowi zarazem jego podstawową słabość. W momencie gdy zachodzi konieczność wprowadzenia nowych kluczy, potrzebna jest metoda ich bezpiecznego przekazania sobie przez partnerów.

> Jak wiemy ze starych filmów wojennych, spotkanie agentów w celu przekazania nowych szyfrów było jednym z najbardziej niebezpiecznych zadań.

Algorytmy szyfrowania symetrycznego: DES,Triple-DES,RC3,RC4,IDEA,Blowfish,AES.

Szyfrowanie asymetryczne

Z powyższego powodu kluczową technologią szyfrowania stosowaną w systemach i sieciach komputerowych jest **szyfrowanie asymetryczne**, zwane również **systemem klucza publicznego**. Klucz szyfrowania każdej jednostki (osoby lub instytucji) składa się z dwóch części: **klucza publicznego**, który jest jawny i może być przesyłany otwartymi kanałami, oraz **klucza prywatnego**, który jest tajny i nigdzie nie wysyłany. Każdy może zaszyfrować wiadomość kluczem publicznym odbiorcy, ale odszyfrować ją będzie mógł tylko właściciel klucza znający jego część prywatną.



Rozwiązuje to podstawową słabość szyfrowania symetrycznego, czyli problem dystrybucji kluczy. Gdy właściciel klucza chce go zmienić, może po prostu wygenerować nową parę kluczy, i dowolnymi kanałami rozpowszechnić nowy klucz publiczny.

Szyfrowanie asymetryczne i symetryczne

Pytanie: czy pojawienie się metod szyfrowania asymetrycznego powinno spowodować, by starsze metody szyfrowania symetrycznego odeszły do lamusa?

Otóż niekoniecznie. Szyfrowanie asymetryczne jest znacznie mniej efektywne (o kilka rzędów wielkości) niż symetryczne. W praktyce, zwłaszcza duże dokumenty, lepiej jest szyfrować tradycyjnymi szyframi symetrycznymi. Można te dwie metody szyfrowania połączyć w taki sposób, że dokument wysyłany danemu odbiorcy jest szyfrowany szyfrem symetrycznym z jednorazowo wygenerowanym kluczem. Ten klucz zostaje zaszyfrowany kluczem publicznym odbiorcy i wysłany mu razem z dokumentem.

Zauważmy, że rozwiązuje to zarazem problem wysyłania jednego dokumentu wielu odbiorcom. Chcąc zaszyfrować go kluczem publicznym, musiałyby zostać utworzone wielokrotne wersje tego samego dokumentu, każda zaszyfrowana kluczem publicznym kolejnego odbiorcy. Zamiast tego, jest jeden dokument zaszyfrowany wspólnym szyfrem symetrycznym, i do niego dołączonych wiele kopii klucza symetrycznego (typowo znacznie mniejszego niż sam dokument) zaszyfrowanego kluczami publicznymi poszczególnych odbiorców.

Popularne algorytmy szyfrowania asymetrycznego: Diffie-Hellman, RSA, ElGamal, i algorytmy krzywych eliptycznych. Uwaga: bezpieczne szyfrowanie algorytmami klucza publicznego wymaga kluczy o długości 1024 bitów lub więcej!

Generacja liczb losowych

Nowoczesna kryptografia potrzebuje liczb losowych do różnych celów takich jak: generacja kluczy, wykorzystanie unikalnych wartości potrzebnych w niektórych protokołach, i innych zastosowań. Co więcej, odporność systemu szyfrowania na ataki zależy od prawdziwej, głębokiej losowości liczb losowych. Niedoskonałość generatora liczb losowych, pozwalająca przewidzieć generowane liczby, zarazem pozwala skonstruować atak na algorytm szyfrowania.

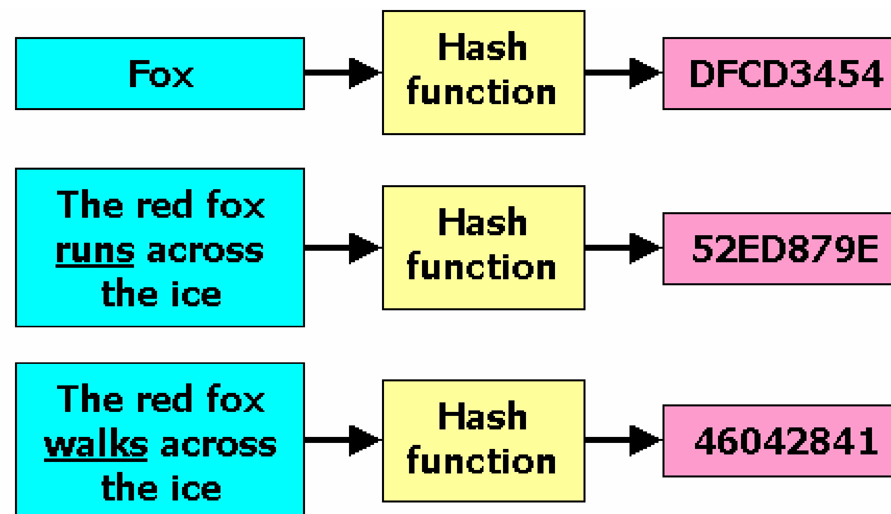
Komputery są urządzeniami deterministycznymi i wygenerowanie liczby prawdziwie przypadkowej jest w nich trudne bądź niemożliwe. W praktyce jednak wystarczy mechanizm generujący liczby nieprzewidywalne i niepowtarzalne.

Jako źródło takich liczb w systemach komputerowych wykorzystuje się różne zjawiska fizyczne takie jak: obarczone szumami elementy elektroniczne, pomiar promieniowania kosmicznego, odbiór szumu radiowego, turbulencje powietrza wewnątrz dysków magnetycznych, pomiar odstępów czasowych napływania pakietów sieciowych, itp.

Gdy potrzebna jest okazjonalna wartość przypadkowa, powyższe źródła mogą jej dostarczyć bezpośrednio. Gdy jednak potrzebna jest większa liczba wartości losowych wykorzystywane są matematyczne generatory liczb pseudolosowych, które generują serię przypadkowych wartości, jednocześnie wykorzystując pojedynczą liczbę losową jako **ziarno** inicjujące sekwencję pseudolosową.

Funkcje mieszające

W systemach informatycznych często przydają się funkcje przypisujące danym argumentom wartości z pewnego zbioru, ale różne dla różnych argumentów. Budowa takich funkcji na ogół polega na silnym mieszaniu różnych fragmentów argumentu (np. bitów), w związku z czym nazywa się je funkcjami **mieszającymi** (*hash functions*).



Własności informatycznych funkcji mieszających: (i) stały, niezależny od argumentu, rozmiar wyniku, (ii) szybkie, deterministyczne obliczanie, (iii) mała liczba kolizji, tzn. jednakowej wartości wyniku dla różnych argumentów. W wielu zastosowaniach informatycznych pewna liczba kolizji jest dopuszczalna, jeśli jest mała w porównaniu z wielkością dziedziny funkcji.

Kryptograficzne funkcje skrótu

Funkcje mieszające znajdują zastosowanie w wielu elementach systemów zabezpieczeń, jednak z tych zastosowań wynikają dodatkowe wymagania dla nich:

- niemożność odtworzenia jakiejkolwiek części argumentu na podstawie wartości funkcji,
- niezwykle małe prawdopodobieństwo (brak praktycznej możliwości) znalezienia innego argumentu, który dałby tę samą wartość funkcji (tzn. kolizji),
- nawet niewielka zmiana argumentu (np. jednego bitu lub znaku) powinna zmienić wartość funkcji do tego stopnia, by niemożliwe/trudne było wykrycie jakiejkolwiek korelacji między tymi wartościami.

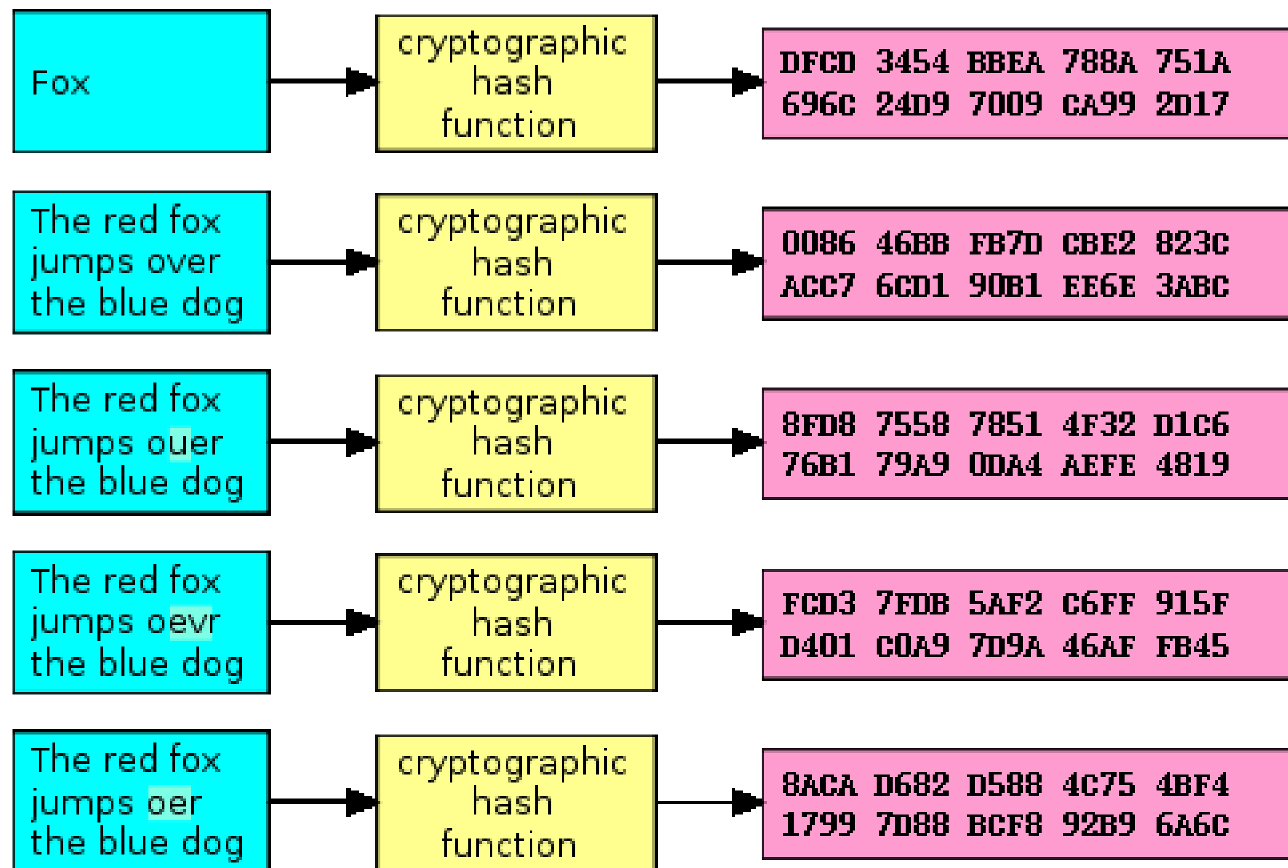
Funkcje mieszające o tych własnościach, stosowane w kryptografii, nazywane są **kryptograficznymi funkcjami skrótu** (*cryptographic hash functions*). Ze względu na wymaganie nieodwracalności stosuje się również określenie funkcji **jednokierunkowych** (*one-way functions*).

Przykładem zastosowania takich funkcji jest przechowywanie haseł. Chcąc zabezpieczyć przechowywane hasła przed możliwością wykradzenia, zamiast przechowywać wersję jawną, możemy przechowywać tylko obliczone skróty, i każdorazowo porównywać je ze skrótem obliczonym z hasła podanego przez użytkownika.

Wkrótce zobaczymy jeszcze inne ważne zastosowanie skrótów kryptograficznych.

Kryptograficzne funkcje skrótu — przykład

Przykład wartości obliczanych przez popularną funkcję SHA-1:



Ze względu na swoje własności, skróty kryptograficzne można traktować jako swojego rodzaju **sygnatury**, niedwuznacznie identyfikujące oryginalny dokument (i jego konkretną, niezniekształconą wersję).

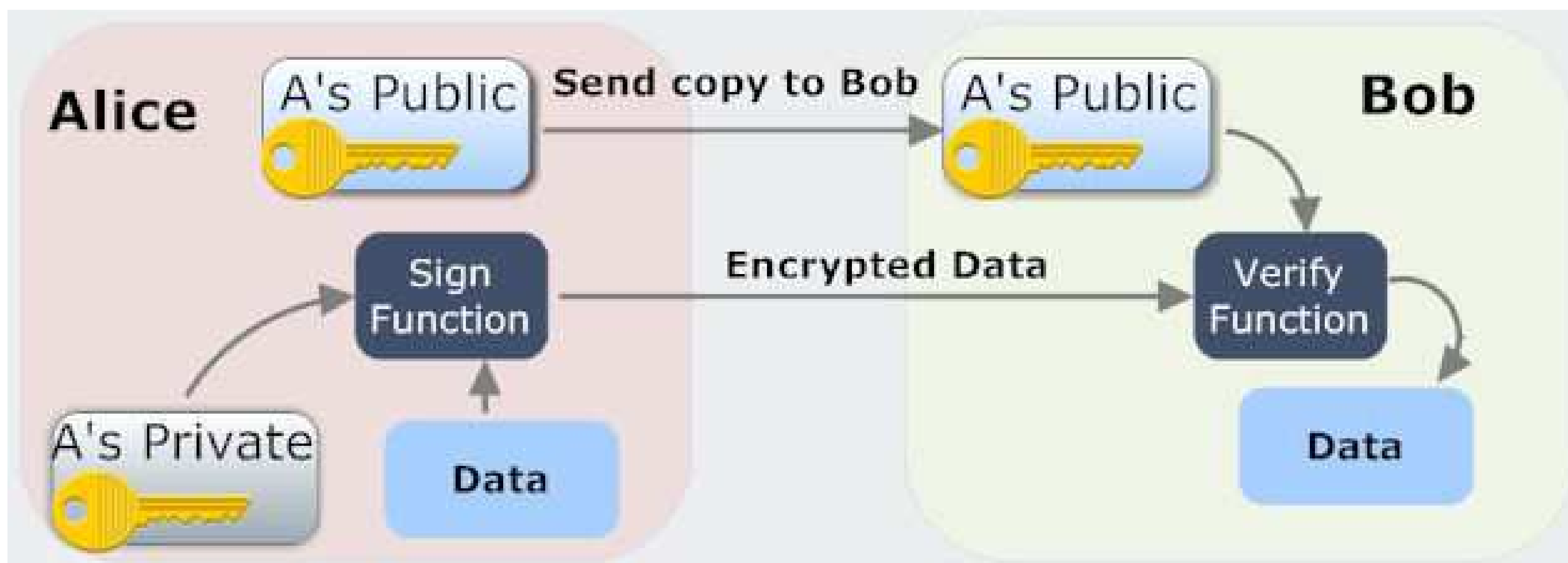
Popularne funkcje skrótów kryptograficznych

Popularny przez wiele lat algorytm skrótu kryptograficznego MD5 generuje ciągi 128-bitowe, często kodowane w postaci 32-znakowych napisów szesnastkowych (heksadecymalnych). Inny popularny algorytm SHA-1 generuje ciągi 160-bitowe, kodowane jako napisy heksadecymalne 40-znakowe.

Jednak kryptografia silnie się rozwija. Istniejące algorytmy są intensywnie badane i poddawane próbom nadużycia, a jednocześnie tworzone są nowe, bezpieczniejsze. Na przykład, w 2011 opublikowano metodę ataku na algorytm SHA-1 pozwalający wygenerować kolizję (alternatywny ciąg bajtów dający tę samą wartość skrótu SHA-1). Metoda wymaga 2^{65} operacji i nikomu nie udało się jeszcze wygenerować takiej kolizji. Pomimo to główni producenci oprogramowania (Microsoft, Google, Mozilla) ogłosili, że od roku 2017 ich systemy nie będą akceptowały certyfikatów opartych na skrótach SHA-1. Istnieje jednak rodzina znacznie bezpieczniejszych algorytmów skrótu SHA-2.

Podpisy cyfrowe

Szyfrowanie kluczem publicznym umożliwia wprowadzenie dodatkowej ważnej funkcji, jaką są **podpisy cyfrowe**. Idea tych podpisów polega na zaszyfrowaniu dokumentu przez nadawcę swoim kluczem prywatnym, i wysłaniu wyniku razem z dokumentem. Zaszyfrowana wersja może być odszyfrowana przez każdego, ale tylko kluczem publicznym nadawcy. Zgodność odszyfrowanego komunikatu z jego pełną wersją dowodzi, że nadawcą jest właściwa osoba, oraz że treść wiadomości jest autentyczna.

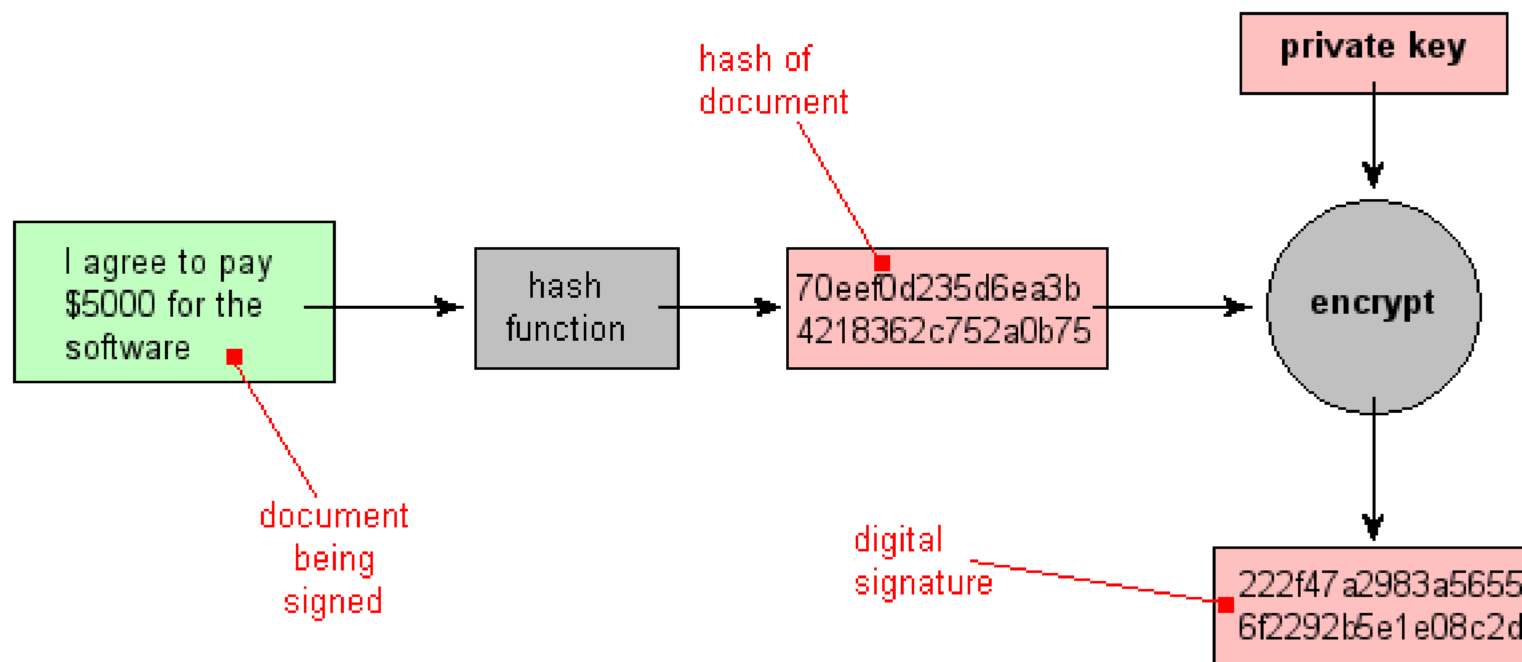


Zauważmy, że aby ta metoda mogła działać, algorytm szyfrowania asymetrycznego musi równie dobrze być w stanie szyfrować kluczem prywatnym i deszyfrować kluczem publicznym, jak i na odwrót. Przykładem algorytmu o tej własności jest RSA.

Skróty kryptograficzne w podpisach cyfrowych

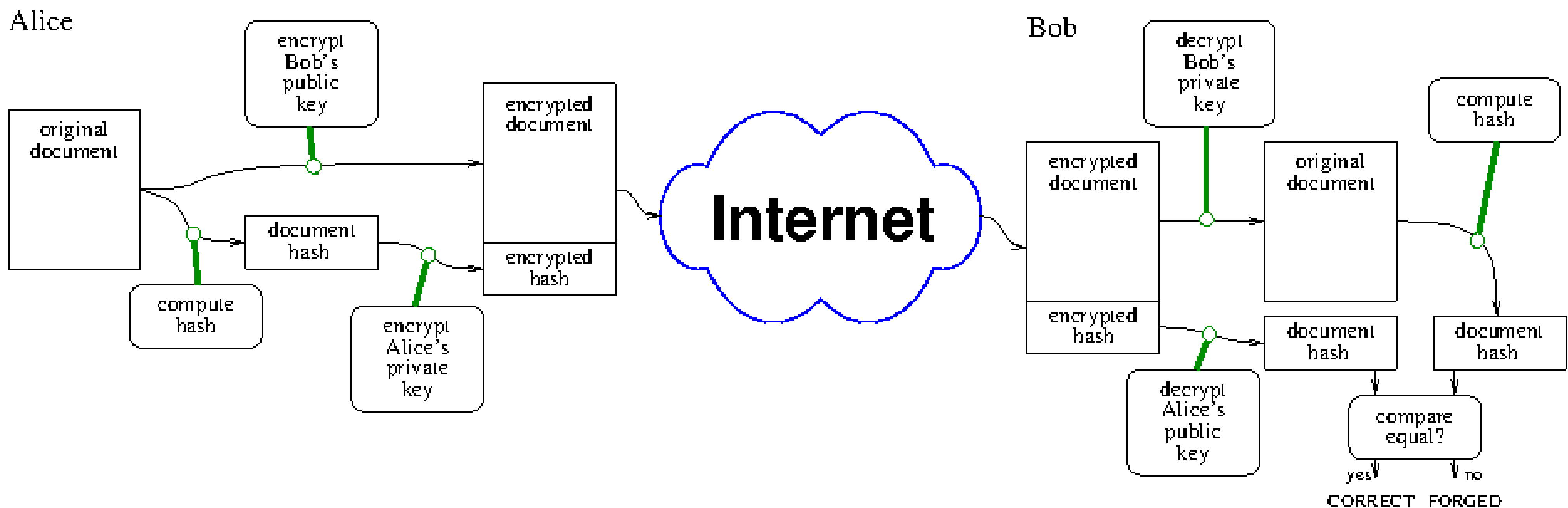
Idea dodania do wysyłanego komunikatu jego zaszyfrowanej wersji jako podpisu działa, ale niekoniecznie jest to wygodne. Na przykład, dla długich komunikatów (takich jak film wideo) podpis byłby niepotrzebnie długi, oraz długo trwałoby jego szyfrowanie.

Dlatego zamiast w roli podpisu cyfrowego szyfrować cały dokument, szyfrujemy w tym celu jego skrót kryptograficzny, który jest krótki, i z definicji prawie jednoznacznie identyfikuje dokument.



Szyfrowanie i podpisywanie

Zastosowanie skrótów kryptograficznych jest zatem standardową i wygodną metodą podpisywania cyfrowego dokumentów. Poniżej przedstawiona jest pełna procedura szyfrowania dokumentu do bezpiecznej transmisji przez sieć, oraz generowania podpisu cyfrowego w celu sprawdzenia integralności dokumentu i wiarygodności jego autorstwa:



Dokładna zgodność obliczonego przez odbiorcę skrótu odebranego dokumentu z rozszyfrowaną wersją otrzymaną świadczy o zgodności dokumentu z wersją wysłaną przez nadawcę, i jednocześnie potwierdza tożsamość nadawcy.

System klucza publicznego

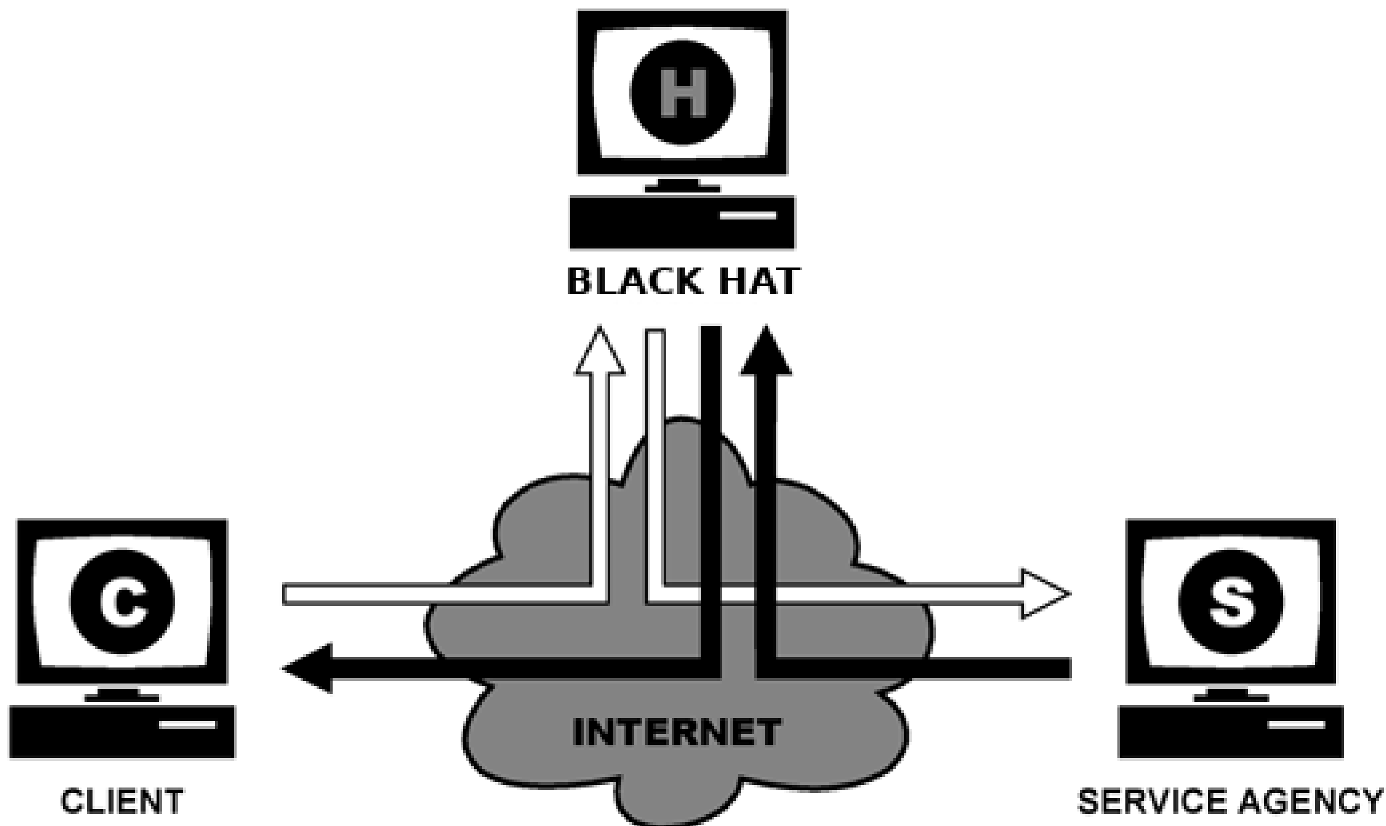
Podsumujmy wiadomości o systemie kluczy publicznych. Pozwala on na zaszyfrowanie komunikatu kluczem publicznym odbiorcy, i jednocześnie wygenerowanie podpisu cyfrowego dokumentu, czyli skrótu kryptograficznego oryginalnego dokumentu zaszyfrowanego kluczem prywatnym nadawcy. Odbiorca może odszyfrować wiadomość swoim kluczem prywatnym, następnie obliczyć jej skrót, i porównać go z otrzymanym od nadawcy skrótem, rozszyfrowanym kluczem publicznym nadawcy. W ten sposób oryginalna wiadomość jest bezpiecznie przesłana w postaci zakodowanej, i odbiorca ma jednocześnie gwarancję, że odebrana wiadomość jest identyczna z wersją nadaną.

Teoretycznie technologia klucza publicznego rozwiązuje problem dystrybucji kluczy szyfrowania. Klucze można przesyłać jawnie otwartymi kanałami. Każdy może np. opublikować swój klucz na stronie internetowej, albo rozsyłać go elektronicznie bez obawy naruszenia poufności danych.

Jednak w masowym użyciu, z jakim mamy do czynienia we współczesnym Internecie, pojawiają się dodatkowe problemy. Klucze ulegają utraceniu i muszą być sprawnie unieważniane i rozsyłane nowe. Niezawodnie można przesłać klucz publiczny przyjacielowi (lub przyjaciółce), ale jak upewnić się, że klucz publiczny banku, firmy Paypal, albo urzędu skarbowego nie został przekłamany? I co, gdyby tak się stało?

Atak pośrednika

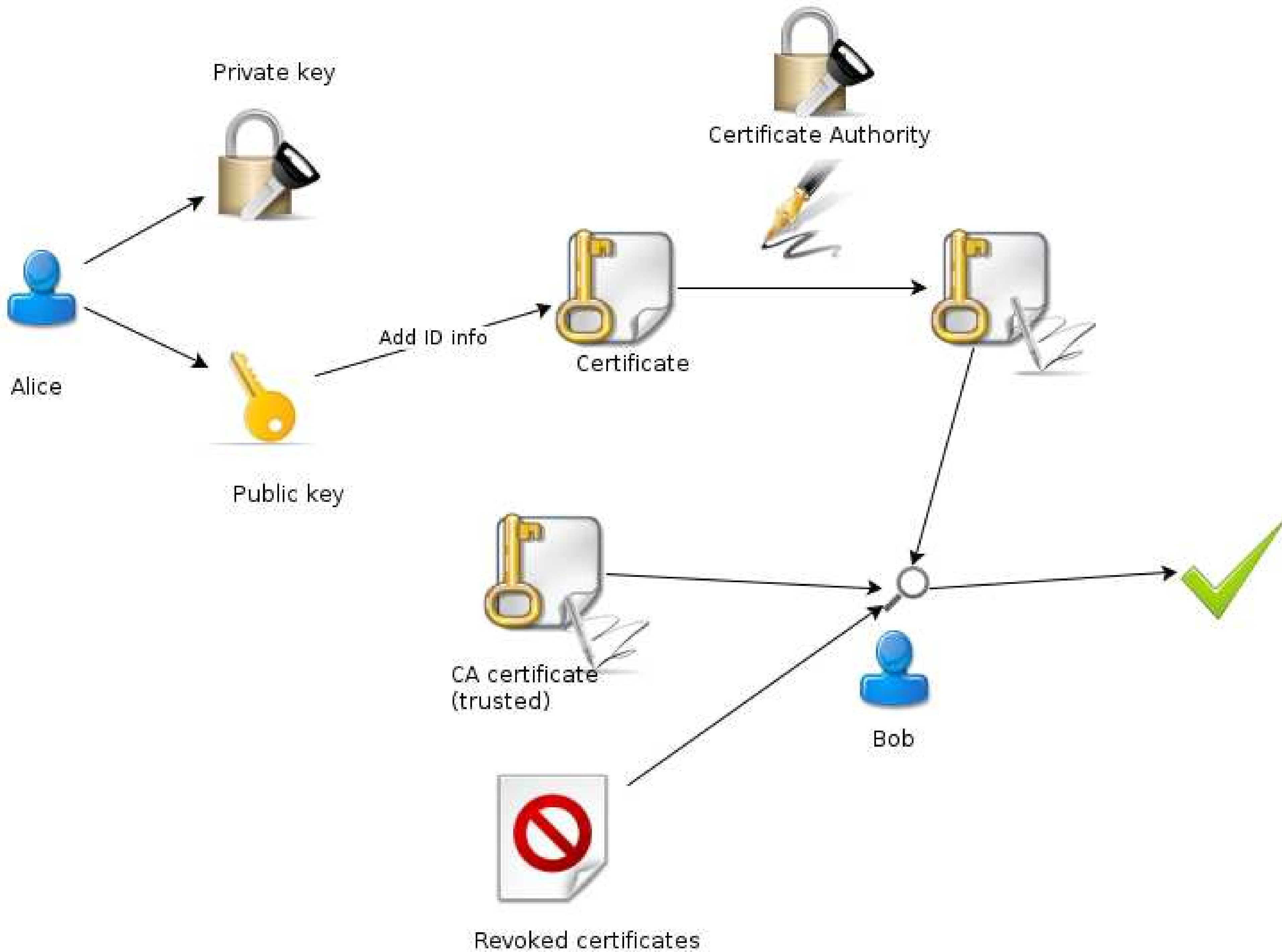
Niestety, w Internecie rozwinęły się liczne techniki ataków wykorzystujące dziury w zabezpieczeniach. Jeżeli komuś uda się przeprowadzić atak w chwili wymiany kluczy publicznych do komunikacji między dwoma partnerami, to może łatwo przechwycić, a nawet sfałszować, całą komunikację między nimi wykorzystując schemat zwany **atakiem pośrednika** (ang. *man-in-the-middle attack*).



Infrastruktura klucza publicznego (PKI)

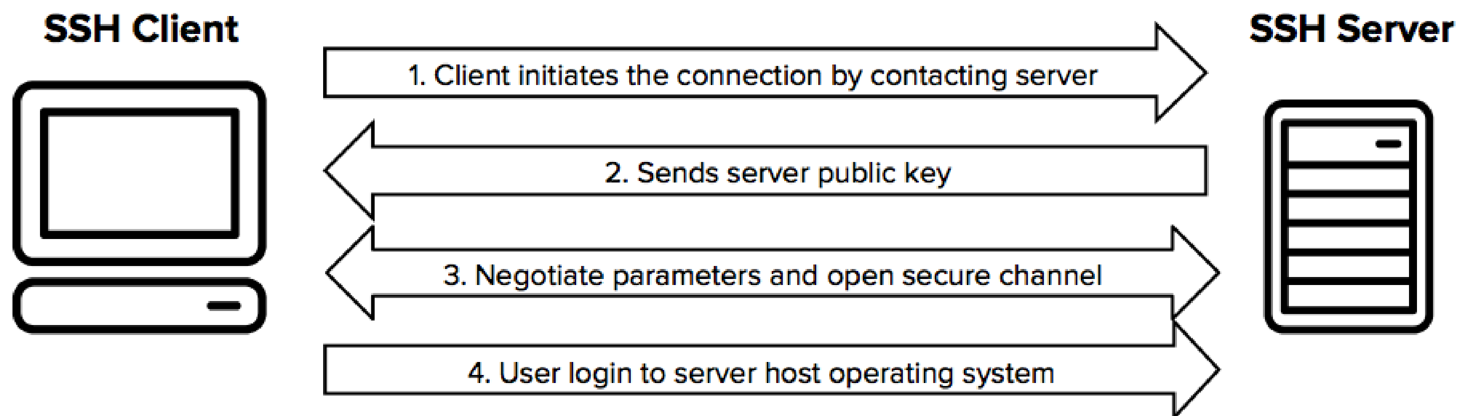
Z powyższych względów system klucza publicznego został w Internecie rozbudowany do **Infrastruktury Klucza Publicznego** (PKI — *Public Key Infrastructure*). Wymaga ona stworzenia zaufanej instytucji zwanej Centrum (albo Urzędem) Certyfikacji CA (*Certification Authority*). Jego rolą jest generowanie **certyfikatów** potwierdzających, że dany klucz publiczny jest rzeczywiście kluczem osoby lub instytucji, która podaje się za właściciela klucza. Ponieważ certyfikat jest podpisany przez CA, więc każdy może sprawdzić, że przesłany klucz publiczny innej jednostki jest właściwy.

Jednocześnie CA przechowuje informacje o unieważnionych kluczach, i pełni szereg dalszych funkcji przydatnych w procesie szyfrowania i podpisywania dokumentów między partnerami w Internecie. Z tego powodu Centrum Certyfikacji musi samo być jednostką w pełni zaufaną, którego ani tożsamość, ani wiarygodność kluczy szyfrowania nie budzą wątpliwości. Na przykład, przeglądarki internetowe mają wbudowane w sobie listy istniejących na świecie CA, i normalnie nie przyjmują certyfikatów podpisanych przez CA spoza tej listy.


Private key
Certificate Authority
Alice
Add ID info
Certificate
Public key
CA certificate
(trusted)
Bob
Revoked certificates

Szyfrowane połączenia ssh

ssh jest programem do logowania się na zdalny system i wykonywania na nim poleceń. Tworzy w tym celu szyfrowany kanał komunikacyjny pomiędzy dwoma niezaufanymi węzłami przez niezabezpieczoną sieć, wykorzystując własny protokół komunikacji.



Połączenie ssh autoryzuje użytkownik posiadający konto na zdalnym systemie. Przed rozpoczęciem tej autoryzacji program ssh sprawdza czy posiada klucz publiczny zdalnego systemu, a gdy go nie posiada, to uzyskuje go przed rozpoczęciem procesu autoryzacji. Klucz publiczny lokalnego systemu jest również wysyłany do zdalnego.

Po nawiązaniu bezpiecznego kanału komunikacyjnego ssh negocjuje ze zdalnym serwerem klucz sesji szyfrowania symetrycznego wykorzystywany w czasie normalnej komunikacji. W czasie tej komunikacji okresowo jest generowany nowy klucz sesji.

Kopiowanie plików programem `scp`

Istnieje program `scp`, który wykorzystuje szyfrowanie i autoryzację `ssh` do kopiowania plików między systemami z wykorzystaniem szyfrowanego kanału.

Istnieje również inny program do szyfrowanego kopiowania plików wykorzystujący połączenie `ssh` o nazwie `sftp`. Ma podobny interfejs to bardzo starego programu kopiowania plików `FTP`, jednak nie wykorzystuje protokołu FTP ani nie wymaga serwera FTP na zdalnym systemie.

Oryginalny program `FTP` przesyłał dane autoryzacyjne użytkownika otwartym, nieszyfrowanym kanałem, podobnie jak program `telnet` służący do tworzenia zdalnej sesji terminalowej. Żaden z tych programów nie powinien być używany we współczesnym Internecie. Nawet jednorazowe użycie programu tego typu może spowodować katastrofalne w skutkach przechwycenie i/lub ujawnienie hasła użytkownika.

Metody autoryzacji ssh

Program ssh dopuszcza szereg metod autoryzacji użytkownika na zdalnym systemie:

login i hasło użytkownika

Autoryzacja przez nazwę i hasło użytkownika jest najprostszą formą, ale niekoniecznie najlepszą. Każdorazowe wpisywanie hasła jest niewygodne, skłania użytkowników do wybierania trywialnych haseł, a przy wielokrotnym użyciu może doprowadzić do ujawnienia hasła przez nieuwagę, przez zastawione pułapki, itp.

pliki .rhosts, .shosts, ...

To jest pradawna metoda autoryzacji pozwalająca użytkownikowi założyć plik i w nim wpisać adresy serwerów, z których powinno być możliwe wejście na konto użytkownika bez żadnej dalszej autoryzacji. Metoda jest uważana za niebezpieczną ze względu na nadużywanie przez nieostrożnych użytkowników, i z tego powodu często jest zablokowana w nowoczesnych systemach.

autoryzowany klucz publiczny

To jest dobra i względnie bezpieczna metoda autoryzacji bez podawania hasła. Zamiast tego, należy na zdalnym systemie wgrać klucz publiczny identyfikatora użytkownika, co umożliwia wykorzystanie klucza prywatnego w procesie logowania.

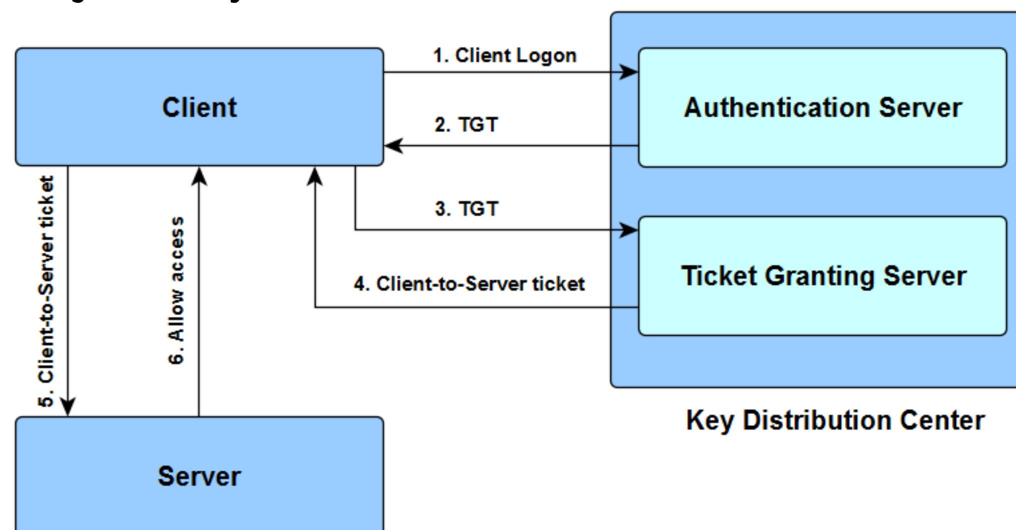
keyboard-interactive

Jest to ogólny schemat autoryzacji użytkownika przez zadawanie mu pytań i sprawdzanie poprawności odpowiedzi. Login i hasło mogą być wariantem tej formy, ale możliwe są inne, np. kod z tokena RSA SecurID. Skonfigurowanie tego typu wariantów autoryzacji jest możliwe przez administratora zdalnego systemu.

inne mechanizmy autoryzacji

- GSSAPI (*Generic Security Service Application Programming Interface*), czyli standardowy interfejs funkcyjny do usług zabezpieczeń, pozwalający oddzielić usługodawcę systemu zabezpieczeń od usługobiorcy. Popularnym systemem zabezpieczeń, który może być wykorzystywany przez GSSAPI jest Kerberos.
- Kerberos jest protokołem uwierzytelniania i autoryzacji w sieci komputerowej z zastosowaniem centrum dystrybucji kluczy KDC.

  Zaprojektowany oryginalnie w latach 80-tych XX wieku w Massachusetts Institute of Technology (MIT). Służy głównie do uwierzytelniania transakcji w systemach klient-serwer z pomocą zaufanego serwera.



Nawiązywanie połączenia ssh

Normalnie przy nawiązywaniu połączeń ze zdalnymi systemami program ssh lub scp zaczyna od sprawdzenia zgodności klucza publicznego zdalnego systemu:

```
shasta-3184> scp -p ~/mozilla.ps sierra:/tmp
The authenticity of host 'sierra (172.16.0.1)' can't be established.
RSA key fingerprint is 1b:9d:e6:cd:df:fd:ac:2a:6c:40:bd:0b:40:2b:50:9d.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'sierra,172.16.0.1' (RSA) to the list of known hosts.
witold@sierra's password:
mozilla.ps           100% |*****************************|   937 KB    00:01
```

W tym przypadku okazało się, że program ssh na komputerze lokalnym (shasta) nie miał zapamiętanego klucza publicznego komputera zdalnego (sierra), i ostrzegł o tym użytkownika, wyświetlając „odcisk palca" otrzymanego klucza publicznego zdalnego serwera. Następnie zapisał otrzymany zdalny klucz w pliku lokalnym na przyszłość. Normalnie jest bezpieczniej użyć od razu klucza publicznego zdalnego komputera nawet do nawiązania wstępnego połączenia, ze względu na możliwość podszywania się.

Przy kolejnych połączeniach system znajdzie i użyje zapamiętany klucz publiczny zdalnego komputera. Warto zwrócić uwagę na ten szczegół, ponieważ jest to pierwszy stopień zabezpieczenia programu ssh.

„Odcisk palca” klucza

Aby potwierdzić poprawność klucza zdalnego systemu, do którego nie mamy innego dostępu niż połączenie `ssh` na swoje konto, możemy sprawdzić jego „odcisk palca” (*fingerprint*), który jest rodzajem sumy kontrolnej, łatwej do przeczytania. Inny użytkownik zalogowany do zdalnego systemu (np. administrator) może go uzyskać następującym poleceniem, i następnie np. podyktować przez telefon:[1]

```
ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
```

Do uzyskania odcisku palca klucza nie jest potrzebny dostęp do klucza prywatnego, do którego zwykły użytkownik nie ma dostępu, ponieważ jest to klucz całego systemu, a nie konkretnego użytkownika.

Po weryfikacji poprawności nowego klucza, należy usunąć stary klucz zdalnego systemu z pliku pamiętanych kluczy, co pozwoli przy kolejnej próbie połączenia zastąpić go nowym kluczem i nawiązać poprawne połączenie:

```
ssh-keygen -R sierra.ict.pwr.wroc.pl
```

[1] Weryfikacja danych za pomocą bezpośredniej rozmowy przez telefon może nie być już bezpiecznym sposobem potwierdzenia wiarygodności. Stworzone za pomocą sztucznej inteligencji boty potrafią podszyć się w rozmowie telefonicznej pod znajomą osobę — naśladując jej głos — w celu przeprowadzenia oszustwa:
https://www.wsj.com/articles/fraudsters-use-ai-to-mimic-ceos-voice-in-unusual-cybercrime-case-11567157402
https://thenextweb.com/security/2019/09/02/fraudsters-deepfake-ceos-voice-to-trick-manager-into-transferring-243000/

Nawiązywanie połączenia ssh (2)

Próba połączenia ssh z systemem, do którego lokalny ssh ma już zapamiętany klucz publiczny może wyświetlić następujący komunikat:

```
-bash-3.00$ ssh sierra.ict.pwr.wroc.pl
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
1b:9d:e6:cd:df:fd:ac:2a:6c:40:bd:0b:40:2b:50:9d.
Please contact your system administrator.
Add correct host key in /home/witold/.ssh/known_hosts to get rid of this message.
Offending key in /home/witold/.ssh/known_hosts:3

RSA host key for sierra.ict.pwr.wroc.pl has changed and you have requested strict
Host key verification failed.
```

W tym przypadku program ssh podniósł alarm, ponieważ klucz przysłany przez zdalny serwer nie zgadza się z jego kluczem zapamiętanym. Może to być wynikiem zmiany klucza na zdalnym serwerze (np. po reinstalacji), ale może sygnalizować fałszerstwo.

Szyfrowane połączenia ssh — generowanie kluczy

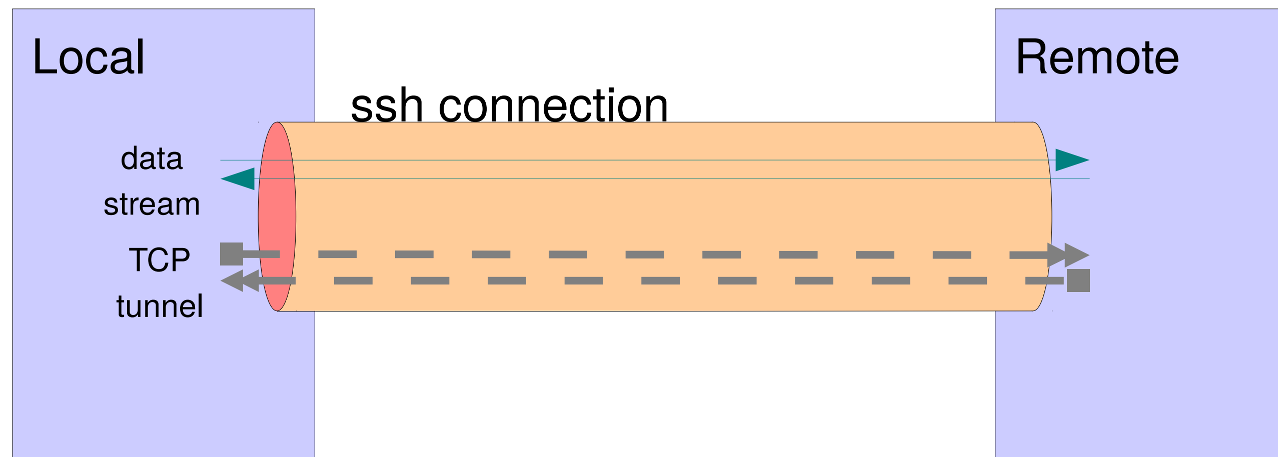
Nawiązywanie połączenia między komputerami protokołem ssh wymaga posiadania klucza publicznego i prywatnego przez każdy z systemów. Ponadto, każdy użytkownik musi mieć wygenerowaną własną parę kluczy:

```
> ssh-keygen
...
> ls -l ~/.ssh:
total 36
-rw-------   1 witold gurus    826 2007-11-25 08:50 authorized_keys
-rw-r--r--   1 witold gurus     15 2007-11-25 10:47 config
-rw-------   1 witold gurus   1675 2007-11-25 08:46 id_rsa
-rw-r--r--   1 witold gurus    396 2007-11-25 08:46 id_rsa.pub
-rw-------   1 witold gurus  14160 2008-10-24 10:16 known_hosts
```

Plik z kluczem prywatnym może być zabezpieczony hasłem (*passphrase*). Hasło trzeba podawać każdorazowo przy korzystaniu z klucza prywatnego (pewne udogodnienie zapewnia program ssh-agent). Można zrezygnować z hasła, wtedy klucz jest chroniony tylko prawami dostępu do pliku.

Tunelowanie połączeń przez ssh

Niezwykle przydatną własnością programu `ssh` jest zdolność **tunelowania połączeń**, czyli tworzenia portów sieciowych TCP po dowolnej stronie połączenia `ssh`. W chwili próby otwarcia tego portu program `ssh` tworzy port po drugiej stronie połączenia `ssh`, i połączenie TCP między nimi. Nawiązywanie połączenia i komunikacja TCP przez łącze `ssh` są szyfrowane.



Mechanizm tworzenia tuneli TCP jest „mocny" i daje wiele możliwości. Przeanalizujemy kilka przykładowych konfiguracji.

ssh: przykłady tuneli (1)

Na przykład, chcemy komunikować się ze zdalnym serwerem HTTP, który może wymagać od nas wprowadzenia danych (przez formularze), które niekoniecznie chcielibyśmy przesyłać przez sieć otwartym tekstem. Serwery HTTP często chronią takie dane udostępniając połączenia szyfrowane HTTPS. Jeśli jednak dany serwer tego nie robi, możemy sami otworzyć swoje połączenie w szyfrowanym kanale. W tym celu należy zainstalować tunel ssh według poniższego schematu:

```
ssh -N -L 8080:localhost:80 remote.ssh.server
```

Komunikacja ze zdalnym serwerem polega teraz na łączeniu się z portem 8080 na komputerze lokalnym. Dodatkowy argument -N mówi, żeby program ssh w ogóle nie otwierał połączenia terminalowego, i obsługiwał tylko tunele TCP.

ssh: przykłady tuneli (2)

W powyższym przykładzie tworzone jest połączenie ze zdalnym serwerem ssh aby umożliwić otwieranie w jego ramach połączeń do innych serwisów na tej samej maszynie. Rozważmy teraz scenariusz, że serwer `ssh` znajduje się w sieci, gdzie istnieją serwisy na innych komputerach, dostępne w sieci wewnętrznej, ale nie bezpośrednio, w Internecie. Może to wynikać z konfiguracji bramy do tej sieci, albo ze względów praktycznych.

Typowo tak są skonfigurowane serwisy, które wymagają autoryzacji otwartym tekstem, na przykład: POP3 (110), Microsoft Windows (139), albo serwisy prywatne, których nie chciano udostępnić w Internecie, tylko w sieci lokalnej, np.: CVS (port 2401), HTTP proxy (2138), i inne.

Poniższe wywołanie `ssh` otworzy połączenie, a w nim cztery tunele dla połączeń TCP do różnych serwisów, na różnych serwerach, widocznych i dostępnych ze zdalnego serwera ssh.

```
ssh -N -L 10110:pop3.server:110 -L 10139:windows.server:139 \
       -L 12138:squid.server:2138 -L 12401:cvs.server:2401 remote.host
```

ssh: przykłady tuneli (3)

Wcześniejsze przykłady zakładały tworzenie portów dla nowych połączeń na maszynie lokalnej. Jest to schemat często przydatny, gdy łączymy się ze zdalnym komputerem, np. za zaporą (*firewall*). Zdalny komputer pełni wtedy rolę bramy do sieci wewnętrznej, i jeśli taka brama istnieje, możemy tworzyć wygodne kanały dostępu do serwisów wewnętrznych.

Rozważmy teraz inny scenariusz, kiedy ponownie chcemy uzyskać dostęp do sieci wewnętrznej za zaporą, lecz nie istnieje komputer dostępowy przyjmujący połączenia `ssh` z Internetu. Możemy jednak utworzyć tunele jeśli posiadamy komputer wewnątrz sieci lokalnej, z którego można wykonywać połączenia na zewnątrz. Wtedy połączenie `ssh` wykonane z zewnętrznym serwerem może zainstalować na tym serwerze port pozwalający otwierać połączenie do naszej własnej maszyny, a za jej pośrednictwem do innych serwisów w sieci lokalnej.

Poniższe wywołanie utworzy tunele dla tych samych serwisów co w poprzednim przykładzie. W tym wypadku serwisy muszą być widoczne z naszego komputera, a porty o numerach > 10000 zostaną utworzone na zdalnej maszynie.

```
ssh -N -R 10110:pop3.server:110 -R 10139:windows.server:139 \
       -R 12138:squid.server:2138 -R 12401:cvs.server:2401 remote.host
```

ssh: przykłady tuneli (4)

Rozważmy teraz sytuację, kiedy chcemy nawiązać połączenie ssh pomiędzy komputerami, pomiędzy którymi nie istnieje możliwość bezpośredniego połączenia. Możemy wtedy zbudować tunele przez kolejne połączenia (klient->brama1->brama2->serwer), które zapewnią otwieranie zagregowanego połączenia ssh:

```
# polaczenie z "klient" przez "brama1" i "brama2" do "serwer"
# na klient:
ssh -L 12345:localhost:12345 brama1
# na brama1:
ssh -L 12345:localhost:12345 brama2
# na brama2:
ssh -L 12345:localhost:22    serwer
```

Efekt jest taki, że wywołanie: `ssh -p 12345 localhost` na komputerze "klient" otwiera połączenie ssh do komputera końcowego "serwer". W celu kopiowania plików pomiędzy tymi komputerami należy wywołać polecenie `scp -P 12345 ...` (w tym przypadku niestandardowy port podajemy przez argument duże „P”).

ssh: przykłady tuneli (5)

W poprzednim przykładzie zestaw tuneli `ssh` pozwolił nawiązywać połączenia `ssh` pomiędzy docelowymi komputerami, gdzie bezpośrednie połączenie nie było możliwe. Rozważmy teraz trochę inny wariant tej sytuacji. Chcemy połączyć komputery klient1 i klient2, które znajdują się w sieciach lokalnych, niedostępnych z Internetu, za pomocą komputera serwer dostępnego w Internecie. Musimy tworzyć tunele od strony izolowanych klientów:

```
# polaczenie z "klient1" przez "serwer" do "klient2"
# na klient1:
ssh -L 12345:localhost:12345 serwer
# na klient2:
ssh -R 12345:localhost:22    serwer
```

Powyższe tunele umożliwiają połączenia `ssh` i `scp` z klient1 na klient2, oczywiście tylko dopóty, dopóki oba połączenia istnieją. Aby umożliwić nawiązywanie połączeń między tymi klientami w odwrotnym kierunku, należy dodać odpowiedni komplet tuneli biegnących w drugą stronę.

ssh: przykłady tuneli (5a)

Rozważmy ponownie dokładnie ten sam scenariusz, co w poprzednim przykładzie. Chcemy połączyć komputery klient1 i klient2, w niedostępnych sieciach lokalnych, za pomocą komputera serwer dostępnego w Internecie. Jesteśmy na komputerze klient1, i jeszcze nie utworzyliśmy tunelu na tej maszynie. Wcześniej, na komputerze klient2 utworzyliśmy odpowiedni tunel:

```
# na klient2:
ssh -R 12345:localhost:22    serwer
```

Na pozór, zamiast tworzyć tunel do portu 12345, i potem łączyć się z końcówką tego tunelu na localhost, moglibyśmy bezpośrednio łączyć się z tym portem na serwer. Tak rzeczywiście jest, ale wymaga to dwóch dodatkowych operacji.

Po pierwsze, domyślna konfiguracja serwera `sshd` nie pozwala udostępniać tworzonych przez tunele portów na zewnętrznych interfejsach sieciowych. Aby tak się stało, należy ustawić flagę `"GatewayPorts yes"` w konfiguracji `sshd`.

Po drugie, operacje tworzenia tuneli (`-L`, `-R`, i `-D`) domyślnie uaktywniają port (bind) tylko na interfejsie wewnętrznym (loopback), pomimo ustawionej powyższej flagi serwera. Aby uaktywnić port również na interfejsach zewnętrznych, należy dodatkowo użyć opcji `-g` w powyższym wywołaniu `ssh`.

SSL i TLS

SSL (Secure Sockets Layer) jest standardem bezpiecznej komunikacji w Internecie. Polega na wbudowaniu technologii szyfrowania danych opartej na infrastrukturze klucza publicznego (Public Key Infrastructure, PKI) w protokół komunikacyjny. Dane są szyfrowane kluczem odbiorcy przed wysłaniem ich. Dzięki temu są zabezpieczone przed możliwością ich odczytania przez nieuprawnioną stronę trzecią, jak również są odporne na manipulację. Jednocześnie system dystrybucji kluczy rozwiązuje problem ich autentyczności.

Bezpieczne połączenia oparte na SSL mogą być stosowane do każdego rodzaju sieciowego protokołu komunikacyjnego, np. HTTP, POP3, FTP, telnet, itd.

SSL nie jest nową technologią. Bieżąca wersja protokołu 3.0 istnieje od 1996. Planowane jest zastąpienie go przez nowszy protokół TLS (Transport Layer Security), który jest podobny ale niekompatybilny. SSL i TLS są najpowszechniej wspieranymi przez serwery WWW szyfrowanymi protokołami (99.8% serwerów wspiera wersję 3.0 SSL, 99.4% wspiera wersję 1.0 TLS).[2]

[2] Stan z kwietnia 2013.

OpenSSL

OpenSSL jest przenośną, wieloplatformową implementacją protokołów SSL i TLS, dostępną na zasadach *open source*. Zasadniczo OpenSSL ma postać biblioteki ANSI C, która implementuje podstawowe operacje kryptograficzne. Poza funkcjami niezbędnymi do szyfrowania sieciowej warstwy transportu, zawiera również funkcje szyfrowania symetrycznego (dla plików), podpisy cyfrowe, kryptograficzne funkcje skrótu, generatory liczb losowych itp.

OpenSSL to więcej niż tylko API, to także program użytkownika z interfejsem wiersza polecenia. Program pozwala na to samo, co API, i dodatkowo, pozwala sprawdzać serwery i klientów SSL.

W tym wykładzie przedstawione są podstawowe możliwości OpenSSL dostępne z narzędzia terminalowego.

Istnieje inna implementacja protokołów SSL/TLS typu open-source — GnuTLS. Zasadnicza różnica między tymi pakietami jest w typie licencji darmowej. Jednak GnuTLS nie posiada narzędzia terminalowego, i z punktu widzenia użytkownika systemów uniksowych jest mało interesujący.

Funkcje programu openssl

Program openssl umożliwia następujące operacje:

- Tworzenie i zarządzanie kluczami prywatnymi, publicznymi i ich parametrami
- Operacje kryptograficzne z kluczem publicznym
- Tworzenie certyfikatów X.509, CSR oraz CRL
- Obliczanie skrótów kryptograficznych wiadomości
- Szyfrowanie i deszyfrowania różnymi szyframi
- Testowanie klientów i serwerów SSL/TLS
- Przetwarzanie poczty podpisanej lub zaszyfrowanej S/MIME
- Żądania znaczników czasowych, generowanie i weryfikacja

Podstawowe wywołania openssl

Program openssl wywołuje się z wektorem argumentów definiującym funkcję:

```
# sprawdzenie zainstalowanej wersji openssl
openssl version
# z obszernymi informacjami
openssl version -a

# lista polecen openssl: jakiekolwiek nieznane polecenie, np.
openssl help
# rowniez opcja -help dla indywidualnych polecen openssl, np.
openssl dgst -help

# kompleksowe testy wydajnosci operacji szyfrowania systemu
openssl speed
# testy wydajnosci ograniczone do konkretnego algorytmu
openssl speed rsa
# testy wydajnosci z uwzglednieniem wieloprocesorowosci
openssl speed -multi 8 rsa
```

Można również wejść w tryb dialogowy openssl i pisać jego polecenia. Jednak brak wtedy możliwości readline — edycji poleceniami Emacsa, historii, itp.

Dostępne algorytmy szyfrowania openssl

```
openssl list -cipher-commands

# lista dostepnych algorytmow z pelna informacja
openssl ciphers -v

# lista tylko szyfrow wersji TLSv1
openssl ciphers -v -tls1

# lista szyfrow "mocnych" (klucze powyzej 128 bitow)
openssl ciphers -v 'HIGH'

# lista szyfrow "mocnych" z algorytmeme AES
openssl ciphers -v 'AES+HIGH'
```

Testowanie zdalnego serwera WWW

Openssl pozwala wykonywać wiele różnych testów zdalnych serwerów HTTPS:

```
# testowanie nawiazywania polaczen ze zdalnym serwerem WWW przez 30s
openssl s_time -connect remote.host:443

# test.polaczen i sciagania strony przez 10 sekund, tworz.nowej sesji
openssl s_time -connect remote.host:443 -www /index.html -time 10 -new
```

Jeśli nie mamy do dyspozycji zdalnego serwera HTTPS, który możnaby wykorzystać do testów, openssl pozwala „postawić” minimalny serwer na wybranym porcie. Serwer serwuje pliki z lokalnego katalogu, w którym został uruchomiony:

```
# uruchomienie serwera HTTPS na porcie 10443 z certyfikatem mycert.pem
openssl s_server -accept 10443 -cert mycert.pem -WWW

# wygenerowanie certyfikatu serwera, zadaje duzo szczegolowych pytan
openssl req \
  -x509 -nodes -days 365 \
  -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

Symetryczne szyfrowanie plików

Szyfrowanie symetryczne jest pomocniczą funkcją `openssl`. W tej roli `openssl` jest mniej wygodny w użyciu niż np. GnuPG. Przy deszyfrowaniu wymaga znajomości, oprócz hasła, algorytmu użytego do szyfrowania.

```
# szyfruj file.txt do file.enc algorytmem AES-256-CBC,
openssl enc -aes-256-cbc -salt -in file.txt -out file.enc
# alternatywna forma tego samego, z kodowaniem tekstowym base64
openssl aes-256-cbc -a -salt -in file.txt -out file.ascii

# deszyfruj plik binarny na stdout
openssl enc -d -aes-256-cbc -in file.enc
# deszyfruj plik zakodowany tekstowo base64 na stdout
openssl enc -d -aes-256-cbc -a -in file.ascii

# haslo szyfrowania mozna podac w wierszu polecenia
openssl enc -aes-256-cbc -salt -in file.txt -out file.enc \
  -pass pass:Kathy123
# mozna rowniez haslo zapisac na pliku
openssl enc -aes-256-cbc -salt -in file.txt -out file.enc \
  -pass file:~/passwords/mypassword.txt
```

```
# koduj tresc pliku base64 (bez szyfrowania), zapisz na drugim pliku
openssl enc -base64 -in file.txt -out file.ascii
# koduj base64 "w locie", wyslij wynik na wyjscie, uwaga na NEWLINE
printf "jakis napis" | openssl enc -base64
# dekodowanie BASE64, ponownie uwaga na NEWLINE
printf "amFraXMgbmFwaXM=\n" | openssl enc -base64 -d
```

Skróty kryptograficzne i podpisy cyfrowe

Skróty kryptograficzne (*file hash* lub *message digest*) pełnią rolę sygnatur dużych plików danych. Na przykład, zamiast porównywać pliki każdy z każdym, można obliczyć ich skróty i szybko je porównać. Jeszcze ważniejszą rolę skróty pełnią przy cyfrowym podpisywaniu przesyłanych danych. Zamiast podpisywać cały plik, co jest równoważne z jego zaszyfrowaniem, można podpisać jego skrót.

Zadaniem skrótu kryptograficznego jest być „prawie unikalnym”. To znaczy, znalezienie innego pliku z tym samym skrótem co dany plik musi być bardzo trudne. W praktyce, podpisanie skrótu jest mniej bezpieczne niż całego pliku.

```
# skrot MD5
openssl dgst -md5 filename
# skrot SHA1
openssl dgst -sha1 filename

# wygenerowanie podpisu skrotu sha1 kluczem prywatnym
openssl dgst -sha1 -sign mykey.pem -out foo.tar.gz.sha1 foo.tar.gz

# weryfikacja sygnatury z kluczem publicznym nadawcy
openssl dgst -sha1 -verify pubkey.pem -signature foo.tar.gz.sha1 \
  foo.tar.gz
```

Przydatne linki

Paul Heinlein, OpenSSL Command-Line HOWTO
http://www.madboa.com/geek/openssl/

J.K. Harris, Understanding SSL/TLS
http://computing.ece.vt.edu/~jkh/Understanding_SSL_TLS.pdf

Philippe Camacho, An Introduction to the OpenSSL command line tool
http://users.dcc.uchile.cl/~pcamacho/tutorial/crypto/openssl/openssl_intro.html

Sun/Oracle documents, Introduction to SSL
http://docs.oracle.com/cd/E19957-01/816-6156-10/