Bezpieczeństwo Systemów i Usług Informatycznych

Laboratorium 5. Reverse_Me

1. Treść zadania.

Naszym zadaniem było przeprowadzenie analizy pliku binarnego w celu uzyskania wygenerowanego przez program hasła dla naszego numeru indeksu, będącego loginem. Następnie należało przeanalizować funkcję generującą hasła dla podanych loginów i napisanie generatora kluczy w dowolnym języku.

2. Odszukanie hasła dla naszego numeru indeksu.

Podczas zajęć zostaliśmy poinstruowani w jaki sposób można przeprowadzić analizę pliku binarnego, w celu odnalezienia hasła. Wykorzystaliśmy do tego zadania program do analizy plików obiektowych objdump oraz debugger gdb. Mając do dyspozycji kod assemblerowy oraz możliwość debugowania programu mogliśmy przejść do linii znajdującej się po wywołaniu funkcji `generate_key()` i odszukać na stosie wygenerowane hasło dla naszego numeru indeksu.

1. `objdump -d Reverse_Me`

```
80487c1:       e8 94 fe ff ff          call   804865a <generate_key>
80487c6:       83 c4 0c                add    $0xc,%esp
```

Rysunek 1. Fragment funkcji Main().

2. `gdb Reverse_Me`

3. `b *0x80487c6...`

4. `x/3x $esp`

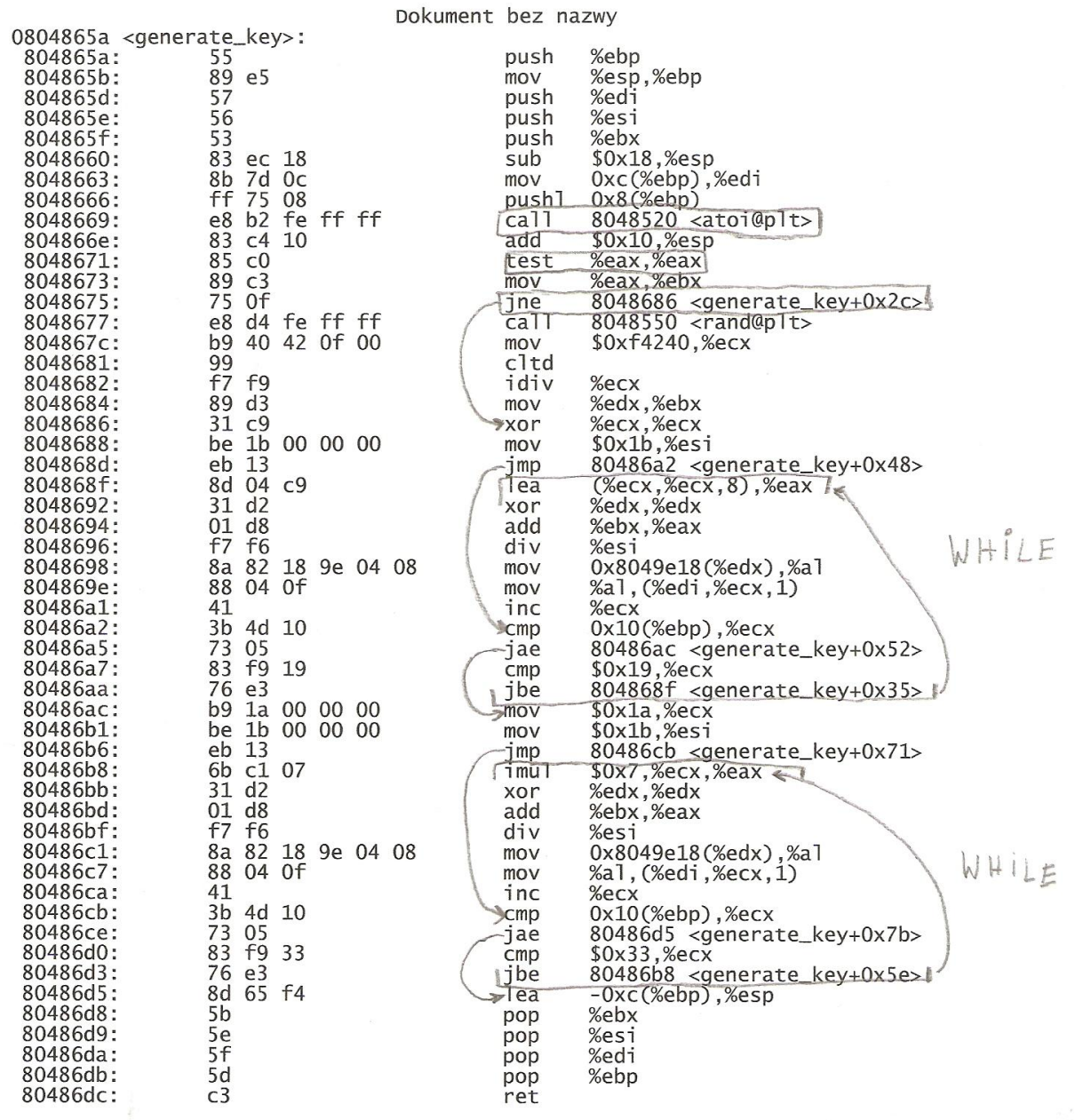
5. `x/s 0x...`

```
Breakpoint 1, 0x080487c6 in main ()
(gdb) x/3x $esp
0xbffff020:     0x0804c008      0x0804c068      0x00000040
(gdb) x/s 0x0804c008
0x804c008:      "209975\n"
(gdb) x/s 0x0804c068
0x804c068:      "xfoxfoxfoxfoxfoxfoxfoxfqxdkryelszfmtOgnuahovbipwc"
(gdb)
```

Rysunek 2. Zrzut ekranu z programu gdb.

3. Utworzenie aplikacji generującej klucze.

Pierwszym krokiem do utworzenia aplikacji generującej klucze jest zrozumienie w jaki sposób powstają. W tym celu należy przeanalizować treść funkcji `generate_key()`. Po raz kolejny przydatny okazał się debugger. Poza wymienionymi w punkcie drugim komendami użyłem komendy "`i r`" wyświetlającej aktualne informacje o rejestrach.



Rysunek 3. Analiza kodu funkcji generate_key().

W kodzie funkcji zaznaczone zostały najważniejsze fragmenty: wywołania funkcji, skoki warunkowe, pętle.

Na początku wywoływania jest funkcja `atoi()` jako argument przyjmuje ona wprowadzany przez nas łańcuch znaków. Następnie próbuje rzutować ten łańcuch na liczbę. Jeżeli się to powiedzie, wykonywany jest skok. Jeżeli się to nie powiedzie, program generuje wartość liczbową w sposób losowy. Jest to powód, ze względu na który nie da się napisać generatora, gdy podana wartość będzie zawierała znaki inne od cyfr.

Następnie napotykamy na dwie pętle. Generują one klucz wynikowy poprzez wykonanie szeregu operacji na wprowadzonej przez nas liczbie. Ciężko jest je opisać słowami, wydaje mi się, że kod C# będzie czytelniejszy.

```
int i = 0;
int j;
int k = 27;
int modulo;
int tmp;
List<char> result = new List<char>();
List<char> alphabet = "abcdefghijklmnopqrstuvwxyz0".ToCharArray().ToList();

while (i < 64 && i <= 25)
{
    j = i + (i * 8);
    j = j + number;
    tmp = j;
    j = j / k;
    modulo = tmp % k;
    result.Add(alphabet[modulo]);

    i = i + 1;
}
```

Listing 1. Pętla pierwsza.

```
i = 26;
k = 27;

while (i < 64 && i <= 51)
{
    j = i * 7;
    j = j + number;
    tmp = j;
    j = j / k;
    modulo = tmp % k;
    result.Add(alphabet[modulo]);

    i = i + 1;
}

Console.WriteLine("KEY: {0}", new string(result.ToArray()));
Console.ReadLine();
```

Listing 2. Pętla druga.

4. Podsumowanie.

Ćwiczenie to pozwoliło nam na opanowanie podstawowych umiejętności analizy kodu po dezasemblacji oraz było powtórką wiadomości dotyczących assemblera. Uświadomiło nam również jakie możliwości kryje za sobą inżynieria wsteczna.