Programowanie Defensywne

Programowanie defensywne i ofensywne
część druga

Arkadiusz Chrobot

Katedra Systemów Informatycznych

14 czerwca 2024

Plan

1. Wprowadzenie
2. Zarządzanie pamięcią
3. Łańcuchy formatujące
4. Arytmetyka zmiennoprzecinkowa
5. Postać kanoniczna ścieżek
6. Wątki
7. Podsumowanie
8. Literatura

Wprowadzenie

W ramach wykładu kontynuowany jest przegląd wybranych zasad i zaleceń dotyczących programowania defensywnego. Poruszane zagadnienia związane są głównie z językami C i Java. Źródłem opisywanych reguł są standardy SEI CERT (ang. SEI CERT *Coding Standards*), które zostały opracowane dla języków programowania C, C++, Java i Perl, a także dla systemu operacyjnego Android.

Zarządzanie pamięcią

Język C

Wprawdzie temat ten był poruszany na poprzednim wykładzie, ale warto przytoczyć przykład, który przedstawiony jest zarówno w standardach SEI CERT, jak i książce autorstwa K&R. Chodzi w nim o korzystanie ze wskaźnika, po zwolnieniu pamięci, na którą on wskazuje. Fragment kodu zaprezentowany na tym slajdzie nieprawidłowo zwalnia jednokierunkowa listę liniową.

```
#include <stdlib.h>
struct node {
  int value;
  struct node *next;
};

void free_list(struct node *head) {
  for (struct node *p = head; p != NULL; p = p->next) {
    free(p);
  }
}
```

Zarządzanie pamięcią

Język C

Problematyczna jest instrukcja `p->next`, bo jej wykonanie następuje po wywołaniu `free(p)`. Poprawny kod może mieć następującą postać:

```
#include <stdlib.h>

struct node {
  int value;
  struct node *next;
};

void free_list(struct node *head) {
  struct node *q;
  for (struct node *p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
  }
}
```

Zarządzanie pamięcią

Bardziej uniwersalnym i przejrzystym rozwiązaniem jest zastosowanie pętli `while`. Z punktu widzenia bezpieczeństwa taki defekt, czyli użycie wskaźnika po zwolnieniu pamięci przez niego wskazywanej, jest podatnością pozwalającą intruzowi na wykonanie kodu z uprawnieniami programu, w którym ona występuje.

Łańcuchy formatujące

Język C i Java

Łańcuchy (lub ciągi) formatujące są wykorzystywane w wielu językach programowania. Są one obecne w C/C++, Java i Python (operator `%`, metoda `format()`) i we wszystkich tych językach mogą być źródłem podatności. Problem powstaje, jeśli program pozwala użytkownikowi na wprowadzenie takiego ciągu, który następnie interpretuje. Standard SEI CERT zawiera przykład kodu z takim defektem, który przedstawiony jest na następnym slajdzie.

Łańcuchy formatujące

Język C

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

void incorrect_password(const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n";
  size_t len = strlen(user) + sizeof(msg_format);
  char *msg = (char *)malloc(len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fprintf(stderr, msg);
  free(msg);
}
```

Łańcuchy formatujące

Język C

Liczba znaków w loginie jest ograniczona do 256, zatem nie ma zagrożenia przepełnieniem całkowitoliczbowym. Zwróćmy jednak uwagę na wiersz nr 20. Łańcuch wprowadzony przez użytkownika jest bezpośrednio przekazywany do strumienia standardowego wyjścia diagnostycznego. Rozwiązanie bezpieczne jest również znacznie prostsze:

```
#include <stdio.h>

void incorrect_password(const char *user) {
  static const char msg_format[] = "%s cannot be
 ↪  authenticated.\n";
  fprintf(stderr, msg_format, user);
}
```

Łańcuchy formatujące

Język C

Jeszcze prostszy przykład podaje David A. Wheeler w książce „Secure Programming HOWTO” (podrozdział 9.2). Niepoprawne jest przekazanie tablicy zawierającej ciąg znaków do funkcji `printf()` w następujący sposób:

```
printf(string_from_untrusted_user);
```

Poprawnie należy to uczynić następująco:

```
printf("%s", string_from_untrusted_user);
```

W przypadku wymienionej funkcji (i podobnych) niebezpieczną może się okazać także specyfikacja konwersji `%n`, która umieszcza liczbę zapisanych znaków do miejsca w pamięci pod wskazanym jako argument adresem. Nie tylko należy unikać tej specyfikacji, ale także pamiętać, że intruz może ją przekazać we wprowadzanym ciągu znaków w pierwszym z wymienionych przykładów.

Łańcuchy formatujące

Język C

W języku C istnieje możliwość ograniczenia w łańcuchu formatującym liczby znaków zapisywanych do bufora. Jest to szczególnie istotne w przypadku funkcji takich, jak `scanf()` lub `fscanf()` i `sscanf()`. Przykład:

```
#include<stdio.h>

int main(void)
{
        char string[10]={0};
        scanf("%9s",string);
        puts(string);
}
```

W przypadku języka C warto rozważyć użycie innych niż standardowa ▸ bibliotek do obsługi łańcuchów znaków.

Łańcuchy formatujące

Standardy SEI CERT zawierając także przykłady właściwe dla języka Java. Kod ze slajdu nr 13 ma wypisać komunikat informujący użytkownika, że data wydania karty płatniczej nie zgadza się z faktyczną, czyli 1995-05-23. Jego autor założył przy tym, że ta druga data nie powinna być znana użytkownikowi. Niestety, jeśli użytkownik wywoła ten program np. w taki sposób (testowane na powłoce `zsh`):

```
java Format "%1\$te-%1\$tm-%1\$tY"
```

to program usłużnie wypisze ją na ekranie. Aby temu zaradzić należy zmienić argument wywołania metody `format()` np. na taki jak w przykładzie pokazanym na slajdzie nr 14.

Łańcuchy formatujące

Język Java

```
import java.util.*;

class Format {
  static Calendar c = new GregorianCalendar(1995,
↪   GregorianCalendar.MAY, 23);
  public static void main(String[] args) {
    // args[0] should contain the credit card expiration date
    // but might contain %1$tm, %1$te or %1$tY format
↪   specifiers
    System.out.format(
      args[0] + " did not match! HINT: It was issued on
↪   %1$terd of some month\n", c
    );
  }
}
```

Łańcuchy formatujące

Język Java

```
import java.util.*;

class Format {
  static Calendar c =
    new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
  public static void main(String[] args) {
    // args[0] is the credit card expiration date
    // Perform comparison with c,
    // if it doesn't match, print the following line
    System.out.format(
      "%s did not match! HINT: It was issued on %terd of some
↪   month\n",
      args[0], c
    );
  }
}
```

Arytmetyka zmiennoprzecinkowa

Język C i Java

Liczby zmiennoprzecinkowe i związana z nimi arytmetyka nie są łatwymi zagadnieniami, dlatego nietrudno w kodzie je wykorzystującym o defekty, które mogą w sprzyjających temu okolicznościach stać się podatnościami. Problem ten dotyczy praktycznie każdego języka programowania, ponieważ każdy z nich korzysta z tego typu liczb. Aby zrozumieć zagrożenia związane ze stosowaniem liczb zmiennoprzecinkowych należy najpierw zapoznać się z ich reprezentacją i jej najczęściej stosowanym formatem, czyli IEEE-754. Dobrym punktem wyjściowym w tym przypadku są artykuły [1, 2].

Arytmetyka zmiennoprzecinkowa

Język C i Java

Standardy SEI CERT podają regułę, która w przypadku obu języków zaleca *nie stosować* zmiennych typów zmiennoprzecinkowych jako liczników pętli.

Listing 1: Pętla może się nie zatrzymać (C)

```c
void func(void) {
  for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
    /* Loop may not terminate */
  }
}
```

Listing 2: Pętla zawsze się zatrzyma (C)

```c
void func(void) {
  for (size_t count = 1; count <= 10; ++count) {
    float x = 100000000.0f + (count * 1.0f);
    /* Loop iterates exactly 10 times */
  }
}
```

Arytmetyka zmiennoprzecinkowa

Język C i Java

Listing 3: Pętla może się nie zatrzymać (Java)

```
for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
  /* ... */
}
```

Listing 4: Pętla zawsze się zatrzyma (Java)

```
for (int count = 1; count <= 10; count += 1) {
  double x = 100000000.0 + count;
  /* ... */
}
```

W obu przykładach, w których pętla się nie zatrzymuje jej licznik jest zwiększany o zbyt małą wartość, aby jakakolwiek jego zmiana nastąpiła.

Arytmetyka zmiennoprzecinkowa

Java

Problemem dla bezpieczeństwa może też okazać się porównanie liczby zmiennoprzecinkowej z wartością `NaN`. Przykłady ze standardu SEI CERT dla języka Java:

Listing 5: Nieprawidłowe porównanie z `NaN`

```
public class NaNComparison {
  public static void main(String[] args) {
    double x = 0.0;
    double result = Math.cos(1/x); // Returns NaN if
 ↪  input is infinity
    if (result == Double.NaN) { // Comparison is always
 ↪  false!
      System.out.println("result is NaN");
    }
  }
}
```

Arytmetyka zmiennoprzecinkowa

Java

Listing 6: Prawidłowe porównanie z `NaN`

```
public class NaNComparison {
  public static void main(String[] args) {
    double x = 0.0;
    double result = Math.cos(1/x); // Returns NaN when
 ↪  input is infinity
    if (Double.isNaN(result)) {
      System.out.println("result is NaN");
    }
  }
}
```

Arytmetyka zmiennoprzecinkowa

C

W języku C dostępne są makra, po włączeniu nagłówka `math.h`, które mogą być wykorzystane do porównania wartości `NaN` z liczbą zmiennoprzecinkową:

```
int isgreater(x, y);
int isgreaterequal(x, y);
int isless(x, y);
int islessequal(x, y);
int islessgreater(x, y);
int isunordered(x, y);
```

Program należy kompilować z opcją konsolidatora `-lm`. Makro o nazwie `islessgreater` sprawdza, czy `(x)>(y)||(x)<(y)`, a makro `isunordered` sprawdza, czy któryś z argumentów jest `NaN`. Argumentami makr muszą być wartości zmiennoprzecinkowe, czyli takie użycie: `isless(a,1)` jest błędne, a takie: `isless(a,1.0f)` prawidłowe.

Arytmetyka zmiennoprzecinkowa
C

Posługując się podprogramami, które przyjmują jako argumenty lub zwracają liczby zmiennoprzecinkowe warto sprawdzić, czy wartości tych liczb są prawidłowe. Może pomóc w tym tabela zamieszczona w standardzie SEI CERT.

Normalizacja łańcuchów znaków

Java

Wprowadzenie standardu Unicode pozwoliło na obsługiwanie znaków, które występują w praktycznie wszystkich językach naturalnych, ale spowodowało również nowe problemy. Aplikacje, przed zastosowaniem filtrowania niezaufanych ciągów znaków powinny poddać je *normalizacji*. Przykład ze standardu SEI CERT, który wyszukuje w ciągu wejściowym znacznik **<script>**:

```
String s = "\uFE64" + "script" + "\uFE65";

// Normalize
s = Normalizer.normalize(s, Form.NFKC);

// Validate
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  // Found blacklisted tag
  throw new IllegalStateException();
} else {
  // ...
}
```

Postać kanoniczna ścieżek

Język Java

Ścieżki do plików mogą zawierać dowiązania oraz inne elementy, które utrudniają ich walidację. Zatem przed jej wykonaniem należy poddać je *sprowadzeniu do postaci kanonicznej* (ang. *canonicalization*). Przykład w języku Java z SEI CERT:

```
File file = new File("/img/" + args[0]);
if (!isInSecureDir(file)) {
  throw new IllegalArgumentException();
}
String canonicalPath = file.getCanonicalPath();
if (!canonicalPath.equals("/img/java/file1.txt") &&
    !canonicalPath.equals("/img/java/file2.txt")) {
  // Invalid file; handle error
}
FileInputStream fis = new FileInputStream(f);
```

Postać kanoniczna ścieżek

Język C

W przypadku języka C istnieje funkcja `realpath()`, która sprowadza ścieżkę do postaci kanonicznej. Została ona zdefiniowana w standardzie POSIX. Jednakże przed wersją 1-2008 tego standardu była ona źle zaprojektowana, ponieważ wymagała jako drugiego argumentu tablicy znaków, której liczba elementów musiała być z góry znana. Niestety, nie można jej było określić w bezpieczny sposób. Obecna wersja pozwala na to, aby drugi argument miał wartość `NULL`. Rozszerzenie języka C zaproponowane przez fundację GNU oferuje funkcję `canonicalize_file_name()`, która jest łatwiejsza w użyciu. Niemniej należy mieć na uwadze, że nie jest ona standardowa. Co ciekawe, w standardzie SEI CERT sprowadzanie ścieżki do postaci kanonicznej w przypadku języka C jest zaleceniem, nie regułą. Przykład użycia `realpath()` ze wspomnianego standardu znajduje się na następnym slajdzie.

Postać kanoniczna ścieżek

Język C

```c
char *realpath_res = NULL;

/* Verify argv[1] is supplied */

realpath_res = realpath(argv[1], NULL);
if (realpath_res == NULL) {
  /* Handle error */
}

if (!verify_file(realpath_res)) {
  /* Handle error */
}

if (fopen(realpath_res, "w") == NULL) {
  /* Handle error */
}

/* ... */

free(realpath_res);
realpath_res = NULL;
```

Wątki

Język Java

Standard SEI CERT wymaga, aby synchronizacja wątków w języku Java odbywała się w blokach z użyciem obiektów prywatnych i finalnych. Inaczej niezaufany kod, który mógłby korzystać z klasy, gdzie zastosowano mechanizm synchronizacji mógłby przeprowadzić atak DoS. Ilustruje to przykład z następnego slajdu.

Wątki

Język Java

```
public class SomeObject {
  // Locks on the object's monitor
  public synchronized void changeValue() {
    // ...
  }
  public static SomeObject lookup(String name) {
    // ...
  }
}

// Untrusted code
String name = // ...
SomeObject someObject = SomeObject.lookup(name);
if (someObject == null) {
  // ... handle error
}
synchronized (someObject) {
  while (true) {
    // Indefinitely lock someObject
    Thread.sleep(Integer.MAX_VALUE);
  }
}
```

Wątki

Język Java

Aby uniknąć tego problemu należy klasę `SomeObject` zdefiniować następująco:

```
public class SomeObject {
  private final Object lock = new Object(); // private
 ↪  final lock object

  public void changeValue() {
    synchronized (lock) { // Locks on the private
 ↪  Object
      // ...
    }
  }
}
```

Podsumowanie

Przedstawione zasady i zalecenia nie są wszystkim, które zostały zdefiniowane w standardach SEI CERT. Pominięte zostały chociażby te, które związane są z zarządzaniem uprawnieniami. Jak przekonaliśmy się na poprzednim wykładzie inne organizacje oraz firmy także wprowadziły (i czasem opublikowały) swoje standardy. Ciekawe podejście zaproponowała OWASP tworząc przewodnik tworzenia bezpiecznego kodu, który nie jest związany z żadnym konkretnym językiem programowania.

Istnieją języki programowania, które zostały (przynajmniej częściowo) zaprojektowane z myślą o bezpieczeństwie. Należą do nich języki imperatywne takie, jak Ada i Rust oraz języki funkcyjne, jak Erlang, Elixir, Haskell i Clojure. Jednakże opanowanie ich może być trudne. Niektóre, jak Rust oferują w związku z tym ułatwienia, ale skorzystanie z nich powoduje, że kod staje się wtedy niebezpieczny [3].

Literatura

David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". W: *ACM Comput. Surv.* 23.1 (mar. 1991), s. 5–48. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

Vincent Lafage. *Revisiting "What Every Computer Scientist Should Know About Floating-point Arithmetic"*. 2020. arXiv: 2012.02492 [math.NA].

Kelsey R. Fulton i in. "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study". W: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, sierp. 2021, s. 597–616. URL: https://www.usenix.org/conference/soups2021/presentation/fulton.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!