Gem #83: Надёжность на основе типов:

2. Проверка входных данных.

Автор: Yannick Moy, AdaCore

Краткое содержание: Gem Ada #83 - Благодаря надёжной системе типов в Ada довольно удобно проверять в процессе компиляции верификацию таких параметров безопасности, как, например, использование значений, которые не могут быть изменены пользователем там, где это необходимо, или надлежащая проверка данных перед их использованием в уязвимом контексте (например, при возможной попытке взлома путём внедрения SQL кода).

В предыдущем Gem Ada #82 обсуждалась обработка данных, которые потенциально могут быть изменены пользователем. В Gem Ada #83 будет разъяснено, как осуществлять проверку входных данных, поступающих команде SQL. (Пройдя по данной ссылке, можно прочитать весёлый комикс, описывающий взлом вводом SQL-кода: http://xkcd.com/327/)

Давайте начнём…

Проверка входных данных состоит из проверки набора свойств входных данных, которые гарантируют правильность их формирования. Обычно это включает исключение набора неверно сформированных входных данных (черный список) или сопоставление входных данных с исчерпывающим набором хорошо сформированных шаблонов (белый список).

Здесь мы рассмотрим задачу проверки входных данных для включения в команду SQL. Это хорошо известная защита от атак путем внедрения SQL, когда злоумышленник передает специально созданную строку, которая интерпретируется как команда, а не как простая строка при выполнении начальной команды SQL.

Основная идея состоит в том, чтобы определить новый тип `SQL_Input`, производный от типа `String`. Функция `Validate` проверяет правильность проверки входных данных и в противном случае завершается ошибкой. Функция `Valid_String` возвращает необработанные данные внутри проверенной строки следующим образом:

```
package Inputs is

   type SQL_Input is new String;

   function Validate (Input : String) return SQL_Input;

   function Valid_String (Input : SQL_Input) return String;

end Inputs;
```

Реализация `Validate` просто проверяет, что входная строка не содержит опасный символ, прежде чем возвращать его как `SQL_Input`, в то время как `Valid_String` является простым преобразованием типа:

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Strings.Maps;  use Ada.Strings.Maps;

package body Inputs is

   Dangerous_Characters : constant Character_Set := To_Set ("""*^';&><</");

   function Validate (Input : String) return SQL_Input is
```

```
   begin
      if Index (Input, Dangerous_Characters) /= 0 then
         raise Constraint_Error
           with "Invalid input " & Input & " for an SQL query ";
      else
         return SQL_Input (Input);
      end if;
   end Validate;

   function Valid_String (Input : SQL_Input) return String is
   begin
      return String (Input);
   end Valid_String;

end Inputs;
```

Теперь, это не предотвращает будущее использование таких преобразований типов в программе, или вредоносный или непреднамеренный. Чтобы принять меры против таких возможностей, мы должны сделать тип `SQL_Input` частный. Чтобы удостовериться мы самостоятельно непреднамеренно не преобразовываем строку ввода в допустимую в реализации Вводов пакета, мы используем эту возможность сделать `SQL_Input` различаемой записью параметризованный состоянием проверки допустимости.

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package Inputs is

   type SQL_Input (<>) is private;

   function Validate (Input : String) return SQL_Input;

   function Valid_String (Input : SQL_Input) return String;

   function Is_Valid (Input : SQL_Input) return Boolean;

private

   type SQL_Input (Validated : Boolean) is
      record
         case Validated is
            when True =>
               Valid_Input : Unbounded_String;
            when False =>
               Raw_Input   : Unbounded_String;
         end case;
      end record;

end Inputs;
```

Каждый раз, когда мы обращаемся к полю `Valid_Input`, дискриминантная проверка будет выполнена, чтобы гарантировать, что операнд типа `SQL_Input` был проверен. Обратите внимание на использование `Unbounded_String` для типа входного компонента, который является более удобным и гибким, чем использование ограниченной строки.

Обратите внимание в реализации `Validate`, что вместо создания исключения, когда строка не может быть проверена, как в первой реализации, здесь мы создаем соответствующие проверенные или недопустимые входные значения на основе результата проверки на опасные символы. Кроме

того, добавлена функция Is_Valid, позволяющая клиентам запрашивать допустимость значения SQL_Input.

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Strings.Maps;  use Ada.Strings.Maps;

package body Inputs is

   Dangerous_Characters : constant Character_Set := To_Set ("""*^';&><</");

   function Validate (Input : String) return SQL_Input is
      Local_Input : constant Unbounded_String := To_Unbounded_String (Input);
   begin
      if Index (Input, Dangerous_Characters) /= 0 then
         return (Validated   => False,
                 Raw_Input   => Local_Input);
      else
         return (Validated   => True,
                 Valid_Input => Local_Input);
      end if;
   end Validate;

   function Valid_String (Input : SQL_Input) return String is
   begin
      return To_String (Input.Valid_Input);
   end Valid_String;

   function Is_Valid (Input : SQL_Input) return Boolean is
   begin
      return Input.Validated;
   end Is_Valid;

end Inputs;
```

Вот и все! Пока этот интерфейс используется, никакие ошибки не могут привести к неправильному вводу, интерпретируемому как команда, гарантируя, что будущие сопровождающие кода непреднамеренно не смогут вставить несоответствующие преобразования.

Конечно, этот минимальный интерфейс на самом деле не предоставляет ничего, кроме проверки входных данных. Просто имея функцию Is_Valid, чтобы сказать, является ли строка допустимыми входными данными, казалось бы, даст вам такую же функциональность. Однако теперь можно безопасно расширить этот пакет дополнительными возможностями, такими как преобразования допустимых входных данных SQL (например, для оптимизации запросов перед их отправкой в базу данных) или для более быстрого разрешения запросов с помощью локального кэша и т. д. Использование закрытой инкапсуляции гарантирует, что ни один клиентский пакет не изменит допустимость входных данных SQL, которыми вы управляете.

Между прочим, аналогичная, но отчётливая проблема очистки входных данных, при которой, возможно, недопустимые данные преобразуются в то, что известно до использования, может быть обработано таким же образом.

Связанный со статьёй текст программы

Attached Files отсутствуют

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Работа Yannick Moy сосредоточена на анализе исходного кода программного обеспечения, в основном для обнаружения ошибок или проверки свойств безопасности / обеспечения безопасности. Yannick ранее работал в PolySpace (теперь The MathWorks), где он начал проект C ++ Verifier. Затем он поступил в INRIA Research Labs / Orange Labs во Францию для проведения PhD по автоматической модульной проверке статической безопасности для программ на C. Янник присоединился к AdaCore в 2009 году, после короткой стажировки в Microsoft Research.

Yannick Moy имеет степень инженера из Политехнической школы (Ecole Polytechnique), магистра в Стэнфордском университете (Stanford University) и доктора философии из Университета Парижа-Суд (Université Paris-Sud). Он является учёным Siebel.

Last Updated: 10/13/2017
Posted on: 4/5/2010

Обсуждение…