Gem #82: Надежность на основе типов:

1. Обработка данных, значение которых может быть потенциально изменено пользователей (данных типа «tainted»).

Автор: Yannick Moy, AdaCore

Краткое содержание: Gem Ada #82 - Благодаря надежной 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 этой и последующей публикации будут представлены два кратких примера данной практики. В Gem Ada #82 обсуждается обработка данных, которые потенциально могут быть изменены пользователем.

Давайте начнём…

Понятия испорченных данных и надёжных данных обычно относятся к данным, поступающим от пользователя по сравнению с данными, поступающими из приложения. Испорченные данные распространяются подобно вирусу, в том смысле, что любой результат вычисления, где один из операндов испорчен, тоже становится испорченным.

Статические анализаторы различного C/C++ обеспечивают средства проверки для испорченных данных, которые помогают найти ошибки, когда данные пользователя используются для вычисления размера выделения памяти под объекты, чтобы злоумышленник мог использовать их для запуска переполнения буфера, приводящего к атаке с повышением привилегий (EoP) процесса.

В Ada компилятор может обеспечить гарантию, что никакие такие ошибки не были представлены случайно (несмотря на то, что Вы можете все ещё обойти правило, если Вы действительно хотите, например при помощи Unchecked_Conversion или оверлейной процедуры по записи адреса), если различные типы используются для испорченных и доверяемых данных без штрафа во время выполнения. Это может быть сделано со многими типами данных, включая основные типы как целые числа.

Скажем, испорченные данные имеют целочисленный тип. Основная идея состоит в том, чтобы получить доверяемый тип из испорченного, и обеспечить значение функции, чтобы добраться до необработанных данных в доверяемом значении, как следующее:

```
package Taint is

   type Trusted_Value is new Integer;

   function Value (V : Trusted_Value) return Integer;
   pragma Inline(Value);

end Taint;
```

Обратите внимание, что реализация Value - это просто преобразование типа:

```
package body Taint is

   function Value (V : Trusted_Value) return Integer is
   begin
      return Integer (V);
```

```
   end Value;

end Taint;
```

Затем убедитесь, что конфиденциальная программа использует надёжные данные:

```
with Taint; use Taint;

procedure Sensitive (X : Trusted_Value) is
begin
   null; --  Do something sensitive with value X
end Sensitive;
```

Попробуем передать данные От пользователя в чувствительную программу `Sensitive`:

```
with Taint;
with Sensitive;

procedure Bad (Some_Value : Integer) is
begin
   Sensitive (Some_Value);
end Bad;
```

Компилятор возвращает ошибку типа:

```
bad.adb:6:15: expected type "Trusted_Value" defined at taint.ads:3
bad.adb:6:15: found type "Standard.Integer"
```

Теперь, это не препятствует тому, чтобы мы делали полезные вычисления на доверяемых данных так же легко как на испорченных данных, включая инициализацию с литералами, операторами выбора, индексацией массива, и т.д.

```
with Taint; use Taint;
with Sensitive;

procedure Good is
   Max_Value : constant := 100;
   X : Trusted_Value := Max_Value;
begin
   X := X + 1; --  Perform any computations on X
   Sensitive (X);
end Good;
```

Поскольку Trusted_Value является типом, производным от типа tainted (Integer), все операции, разрешенные для данных tainted, также разрешены для доверенных данных, но операции их смешивания запрещены.

Имейте в виду, что ничто не мешает самой программе свободно конвертировать между испорченными данными и доверенными данными, но это требует вставки явного преобразования, которое можно заметить во время просмотра кода.

Чтобы полностью предотвратить такие непреднамеренные преобразования (например, для облегчения обслуживания), Тип, используемый для надежных данных, должен быть частным, чтобы только пакет, который определяет его, мог преобразовываться в него и из него. Поскольку Trusted_Value является частным, мы также должны предоставить соответствующую функцию для каждого литерала, который мы использовали ранее, а также операции, которые мы хотели бы разрешить для доверенных значений (обратите внимание, что для эффективности все операции могут быть встроены):

```
package Taint is

   type Trusted_Value is private;

   function Value (V : Trusted_Value) return Integer;

   function Trusted_1 return Trusted_Value;
   function Trusted_100 return Trusted_Value;

   function "+" (V, W : Trusted_Value) return Trusted_Value;

private

   type Trusted_Value is new Integer;

end Taint;
```

Новая реализация как ожидалось:

```
package body Taint is

   function Value (V : Trusted_Value) return Integer is
   begin
      return Integer (V);
   end Value;

   function Trusted_1 return Trusted_Value is
   begin
      return 1;
   end Trusted_1;

   function Trusted_100 return Trusted_Value is
   begin
      return 100;
   end Trusted_100;

   function "+" (V, W : Trusted_Value) return Trusted_Value is
   begin
      return Trusted_Value (Integer (V) + Integer (W));
   end "+";

end Taint;
```

Конечно, теперь клиент должен быть адаптирован к этому новому интерфейсу:

```
with Taint; use Taint;
with Sensitive;

procedure Good is
   X : Trusted_Value := Trusted_100;
begin
   X := X + Trusted_1; --  Perform any computations on X
   Sensitive (X);
end Good;
```

Это оно! Никакие ошибки не могут привести к тому, что пользовательские данные случайно будут переданы пользователем, где ожидаются надёжные данные, и будущие сторонники кода не будут подвержены соблазну вставлять преобразования, когда компилятор жалуется.

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 Яника Моя 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.

Янник 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: 3/22/2010

Обсуждение…