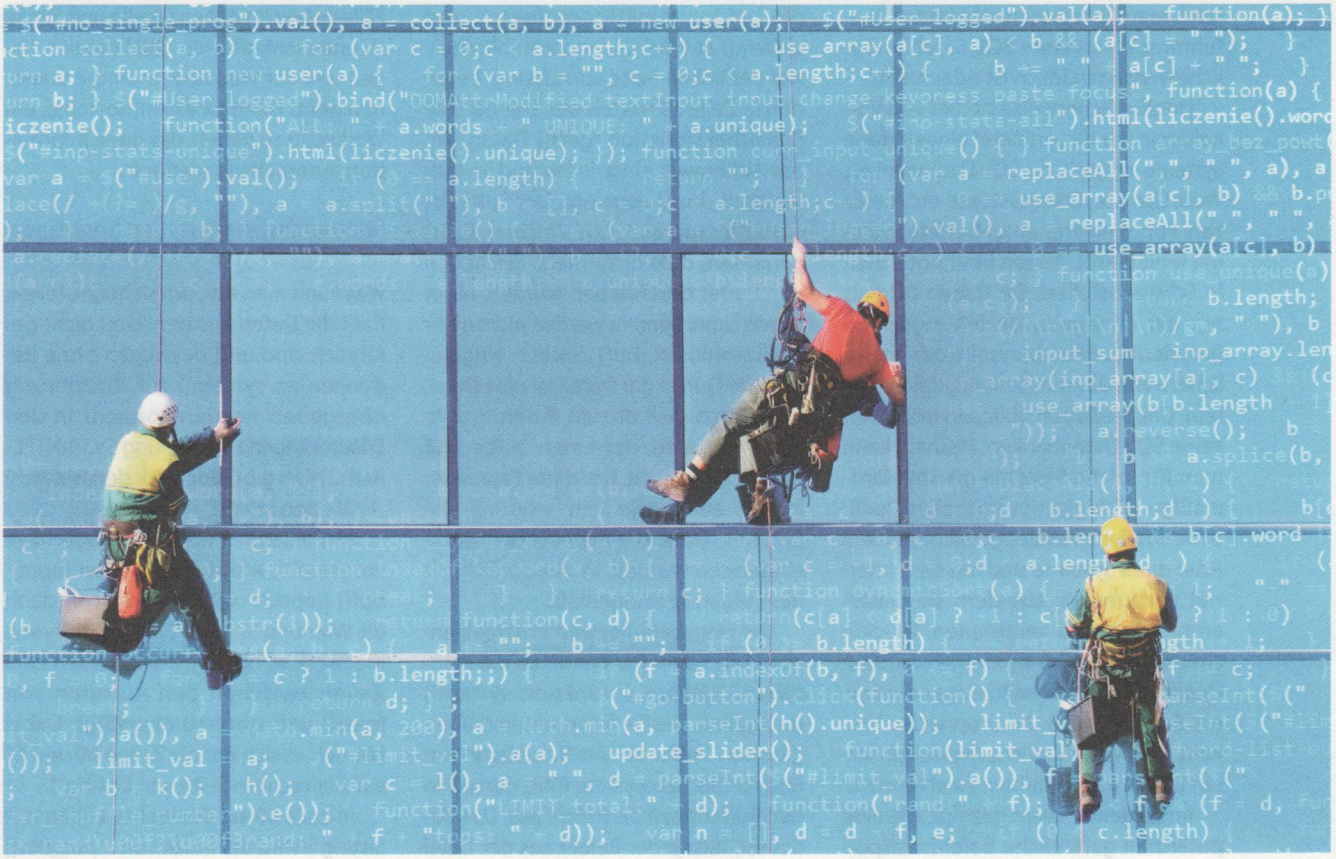
Statische Quellcodeanalyse sorgt für Security:

Reinigung für den Code



(Bild: Shutterstock)

Durch Einlesen externer Daten können schädliche Befehle zur Ausführung gelangen. So lange ein Standard-Betriebssystem verwendet wird, lassen sich die Gefahrenstellen automatisch erkennen. Wenn ein proprietäres oder gar kein Betriebssystem vorhanden ist, muss das Analysewerkzeug entsprechend konfigurierbar sein.

Von Frank Büchner

Bisher stand bei eingebetteten Systemen oftmals der Safety-Aspekt (Schutz des Menschen vor dem System) im Vordergrund. Der Security-Aspekt (Schutz des Systems vor dem Menschen) wurde stiefmütterlich behandelt. Security rückt jedoch verstärkt ins Bewusstsein, insbesondere durch Medienberichte über Hackerangriffe auf Kraftfahrzeuge, Sicherheitslücken in medizinischen Geräten und verschiedenen Bestandteilen des Internet of Things (IoT) wie Webcams, TV-Geräten, Stromzählern und ähnlichem. Für „große" eingebettete Systeme mit Betriebssystemen wie Windows oder Linux gibt es Werkzeuge, die vollautomatisiert gewisse Security-Schwachstellen aufdecken. Diese Verfahren eignen sich aber auch für „kleinere" eingebettete Systeme, die nur ein rudimentäres, möglicherweise proprietäres Betriebssystem haben.

Security-Schwachstellen gibt es in unterschiedlichen Ausprägungen. Es fängt damit an, das IoT-Systeme wie Webcams alle mit dem gleichen Passwort ausgeliefert werden (was den Zugriff recht einfach macht) und endet beim Ausnutzen von Zero-Day-Schwachstellen (was tiefgehende Kenntnisse und Arbeitsaufwand erfordert).

Schädlicher Code

Eine bekannte Security-Schwachstelle ist die Einspeisung von schädlichem (tainted) Code bzw. Daten. Hierzu gehört beispielsweise die Einspeisung von manipulierten Befehlen (Command Injection) in ein System, die dann „offiziell" ausgeführt werden, aber keine offiziell erwünschten Auswirkungen haben. Ein Beispiel hierfür ist SQL-Injektion, wobei ein schädlicher SQL-Befehl für eine Datenbank ausgeführt wird, was im Extremfall zur Löschung der kompletten Datenbank führen kann. Ein anderes Beispiel ist XPath-Injektion, durch welche die Daten einer Web-Applikation offengelegt werden können. Moderne statische Quellcode-Analysewerkzeuge können solche Einspeisungen prinzipiell ohne weiteres Erkennen, denn es ist bekannt, wie und wo ein SQL-Befehl ausgeführt wird. An den fraglichen Stellen im Quellcode

```
59 fgets(input_buf, sizeof(input_buf), fp);  59 fgets(input_buf, sizeof(input_buf), fp);
60 system(input_buf);                        60 strncpy(command, input_buf, BUF_LEN);
61                                           61 sanitize(command);
62                                           62 system(command);
```

Listing 1. Auf der linken Seite kann ein schädlicher Befehl ausgeführt werden; auf der rechten Seite ist dies verhindert.

kann man überprüfen, ob der zur Ausführung kommende SQL-Befehl möglicherweise schädlich ist oder nicht.

Unglücklicherweise ist dies von Vorneherein nur für „größere" eingebettete Systeme möglich, die SQL-Datenbanken, Web-Dienste, Dateisysteme etc. besitzen. Für „kleinere" eingebettete Systeme fehlt dieses Wissen über potenzielle Schwachstellen. Sie haben oftmals nur ein minimales beziehungsweise proprietäres Betriebssystem oder möglicherweise überhaupt kein Betriebssystem. Und sie haben üblicherweise auch keine SQL-Datenbanken. Deshalb kann man für solche Systeme die standardmäßigen Fähigkeiten statischer Quellcode-Analysewerkzeuge nicht nutzen. Aber diese Systeme sind mit dem Internet verbunden, aus dem sie Daten empfangen und verarbeiten. Glücklicherweise können moderne statische Quellcode-Analysewerkzeuge erweitert werden, um auch die proprietäre Verwendung von möglicherweise infizierten Daten bzw. Befehlen zu erkennen.

Im Folgenden wird gezeigt, wie eine potenziell gefährliche Befehlseinspeisung auf einem Standard-Betriebssystem wie Windows aussehen und wie diese Schwachstelle beseitigt werden kann. Danach wird eine vergleichbare Befehlseinspeisung auf einem kleineren eingebetteten System betrachtet, wo naturgemäß proprietäre Eingabefunktionen verwendet werden. Das Aufspüren der Schwachstellen erfolgt im Beispiel mit dem statischen Quellcode-Analysewerkzeug Klocwork.

Standardschwachstellen finden

Als Beispiel für eine von vornherein bekannte Befehlseinspeisung dienen die beiden Funktionen fgets() und system() aus der C-Standardbibliothek. Auf der linken Seite von **Listing 1** werden in Zeile 59 mittels der Standard-C-Bibliotheksfunktion fgets() Daten von der Außenwelt eingelesen und im Puffer input_buf[] abgelegt. Der Puffer ist ein Array mit *BUF_LEN* Zeichen. Es ist garantiert, dass nicht mehr Zeichen in den Puffer geschrieben werden, als er fassen kann, denn es werden nicht mehr als sizeof(input_buf) Zeichen eingelesen. Die Quelle der Daten ist eine Datei, was durch den dritten Parameter *fp* bestimmt wird, der einen Zeiger auf eine Datei darstellt. Der dritte Parameter könnte auch *stdin* sein, wodurch die Zeichen über die Standardeingabe eingelesen würden. In diesem Fall gilt das Folgende sinngemäß.

Der Eingabepuffer wird in der folgenden Zeile 60 als Parameter an die Standard-C-Bibliotheksfunktion system() übergeben. Die Funktion system() übergibt den Puffer an das Betriebssystem und das Betriebssystem führt daraufhin den Pufferinhalt als Befehl aus. Das bedeutet, dass jeder Befehl aus der Datei (deren Herkunft unbekannt ist), vom Betriebssystem ausgeführt wird, egal was der Befehl bewirkt. Das vorliegende Beispiel wurde unter Windows erarbeitet, gilt im Prinzip aber auch für Linux.

Unter Windows wäre es harmlos, das Kommando „dir" auszuführen, aber es kann katastrophal sein, wenn das Kommando „del" ist. Deshalb erzeugt Klocwork bei der Analyse die Warnung SV.TAINTED.INJECTION für den Aufruf von system() in Zeile 60 (**Bild 1**) und zeigt den Weg der möglicherweise schädlichen Daten von der Quelle zur Senke. Im vorliegenden Fall ist dies offensichtlich. Es kann aber durchaus Fälle geben, in denen Quelle und Senke in unterschiedlichen Quelldateien liegen und der Weg über mehrere Stationen geht. Dann ist es komfortabel, ein Werkzeug zu haben, das diese Information automatisch ermittelt.

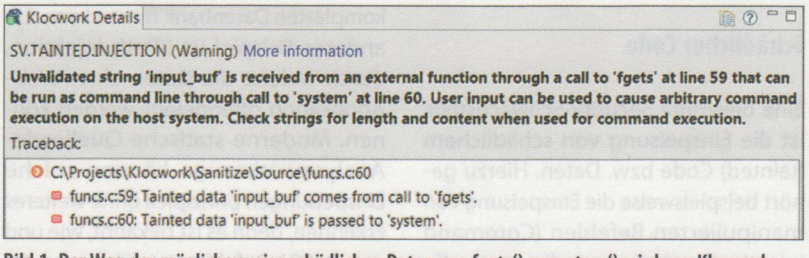


Bild 1. Der Weg der möglicherweise schädlichen Daten von fgets() zu system() wird von Klocwork angezeigt.

Standardschwachstellen beseitigen

Was kann man tun, um sicherzustellen, dass die Daten aus der Datei nicht gefährlich sind und deswegen ohne Bedenken an system() zur Ausführung übergeben werden können? In den Erläuterungen zur Warnung SV.TAINTED.INJECTION gibt Klocwork den Hinweis: „Prüfe den Inhalt!"

Im Folgenden wird eine Möglichkeit vorgestellt, wie der Inhalt von input_buf[] geprüft werden kann, was dann die Warnung SV.TAINTED.INJECTION von Klocwork verhindert. Dazu werden zwei Zeilen zwischen den Aufrufen von fgets() und system() eingefügt. Dabei handelt es sich um die Zeilen 60 und 61 auf der rechten Seite von Listing 1.

In Zeile 60 auf der rechten Seite von Listing 1 wird der Inhalt von input_buf[] in einen neu erzeugten Puffer command[] mit passender Größe und passendem Typ umkopiert. Das Umkopieren alleine garantiert natürlich nicht, dass der Inhalt ungefährlich ist und das Umkopieren wird von Klocwork deshalb nicht als valide Prüfung akzeptiert; die Warnung für den Aufruf von system() würde bestehen bleiben. Die tatsächliche Prüfung erfolgt durch den Aufruf der Funktion sanitize(). Aber wie könnte eine valide Prüfung in sanitize() aussehen?

Es gibt mehrere Möglichkeiten: Die Funktion sanitize() könnte prinzipiell alle Befehle zulassen, aber die Ausführung bestimmter Befehle, beispielsweise „del", verhindern (Blacklisting). Die Funktion sanitize() könnte umgekehrt alle Befehle blockieren und nur die Ausführung bestimmter Befehle zulassen (Whitelisting). Es gibt noch weitere Möglichkeiten, je nachdem, was gewünscht wird. Wenn beispielsweise gewünscht wird, dass nur ein einzelner Befehl ausgeführt wird, könnte man nach dem Zeichen ‚&' in der Eingabe

```
void sanitize(char *command)
{
  char buf[BUF_LEN]="";

  if (0 == strncmp(command, „dir“, 3))
  {
    strcpy(buf, „dir“);
  }
  else if (0 == strncmp(command, „path“, 4))
  {
    strcpy(buf, „path“);
  }
  strncpy(command, buf, strlen(buf)+1);
}
```

Listing 2. Die Funktion sanitize() erlaubt nur zwei Befehle.

```
Read_command_from_Bluetooth(input_buf);
strncpy(command, input_buf, BUF_LEN);
Sanitize_command_from_Bluetooth(command);
Exec_command_from_Bluetooth(command);
```

Listing 3. Für Exec_command_from_Bluetooth() gibt es keine Warnung mehr.

suchen und es selbst und alles, was danach kommt, entfernen. Das Zeichen ‚&' trennt unter Windows zwei aufeinanderfolgende Befehle.

Die Funktion sanitize() in **Listing 2** ist ein Beispiel für Whitelisting. Sie erlaubt im vorliegenden Fall nur die Ausführung der zwei vorgegebenen Befehlen „dir" und „path". Alle anderen Befehle schaffen es nicht durch sanitize() und werden deshalb nicht durch system() ausgeführt. Bei der Arbeit mit Klocwork ist zu beachten, dass ein separater Puffer buf[] als Zwischenspeicher für die zugelassenen Befehle verwendet werden muss. Dieser Puffer buf[] wird dann auf command[] kopiert. Klocwork akzeptiert Blacklisting nicht als Reinigungsmethode.

Proprietäre Schwachstellen finden

Im vorhergehenden Beispiel konnte Klocwork die Schwachstelle finden, weil die Semantik der Funktionen fgets() und system() bekannt ist, da sie aus der Standard-C-Bibliothek stammen. Aber kleinere eingebettete Systeme nutzen möglicherweise proprietäre Funktionen zur Kommunikation mit der Außenwelt, und die Semantik dieser proprietären Funktionen ist nicht bekannt. Als Beispiel ein Paar von zwei Funktionen:

```
Read_command_from_Bluetooth()
Exec_command_from_Bluetooth()
```

Die Funktion Read_command_from_Bluetooth() könnte ein gefährliches Kommando einlesen und an Exec_command_from_Bluetooth() übergeben. Diese Situation ähnelt sehr stark der Situation auf der linken Seite von Listing 1. Allerdings gibt es keine Warnung durch Klocwork. Das liegt daran, dass die Semantik der beiden Funktionsaufrufe unbekannt ist, da die Namen der Funktionen proprietär und willkürlich gewählt sind. Damit Klocwork warnen kann, muss es über die Bedeutung der Funktionen informiert werden.

Das statische Analyse-Werkzeug Klocwork verfügt über eine Wissensdatenbank, die Informationen über wohlbekannte Fakten enthält, beispielsweise die Semantik von fgets() und von system(). Diese Wissensdatenbank kann erweitert werden, und damit kann die Security-Schwachstelle auch bei dem Bluetooth-Befehlspaar automatisch erkannt werden. Zwei Einträge erweitern die Wissensdatenbank von Klocwork:

```
Read_command_from_Bluetooth -
TSsrc $1
Exec_command_from_Bluetooth -
TSSinkInj $1
```

Bei diesen beiden Einträgen steht am Beginn der Zeile der Funktionsname, den der Eintrag betrifft. Das bedeutet, man kann für beliebige Funktionsnamen Information hinzufügen. In der ersten Zeile steht „TSsrc" für „Tainted String Source", was bedeutet, dass es sich bei Read_command_from_Bluetooth() um eine Quelle von möglicherweise schädlichen Daten handelt. In der zweiten Zeile steht „TSSinkInj" für „Tainted String Sink Injection", was angibt, dass Exec_command_from_Bluetooth() eine Datensenke für eine (Befehls-)Injektion ist. In beiden Zeilen gibt $1 an, dass es sich bei den beschriebenen Daten jeweils um den ersten Parameter der Funktionen handelt.

Ist die Wissensdatenbank von Klocwork um die zwei Zeilen erweitert, meldet das Tool bei Aufruf von Read_command_from_Bluetooth analog zu Bild 1 die Warnung SV.TAINTED.INJECTION. Das ist analog zur gleichen Warnung für Zeile #60 auf der linken Seite von Listing 1. Analog zu Bild 1 zeigt Klocwork auch den Weg der möglicherweise schädlichen Daten an.

Proprietäre Schwachstellen beseitigen

Analog zur rechten Seite von Listing 1 wird durch Einfügen von strncpy() und Sanitize_command_from_Bluetooth() die Warnung SV.TAINTED.INJECTION verhindert (siehe **Listing 3**). Der Inhalt der Funktion Sanitize_command_from_Bluetooth() kann analog der Funktion sanitize() aus Listing 2 sein.

Statische Code-Analysewerkzeuge können die Ausführung von möglicherweise gefährlichen Befehlen erkennen. Das ist ohne weiteres möglich, falls die Anwendung Standardmechanismen zum Einlesen und Ausführen der Befehle verwendet. Wenn proprietäre Mechanismen verwendet werden, können fortgeschrittene Werkzeuge erweitert werden, um auch Security-Schwachstellen durch proprietäre Mechanismen zu erkennen. In jedem Fall erspart die automatisierte Erkennung großen Aufwand gegenüber der manuellen Prüfung. *jk*

Literatur

[1] http://www.hitex.de/klocwork: More about the static software analysis tool Klocwork

[2] Seacord, Robert C., Secure Coding in C and C++, second edition 2013, Addison-Wesley.

Frank Büchner
hat ein Diplom in Informatik von der Technischen Hochschule Karlsruhe. Seit mehreren Jahren widmet er sich dem Thema Testen und Softwarequalität. Seine Kenntnisse vermittelt er regelmäßig durch Vorträge und Fachartikel. Momentan arbeitet er als „Principal Engineer Software Quality" beim Unternehmen Hitex in Karlsruhe.
Frank.Buechner@hitex.de