world of solutions

Elektronik

SONDERDRUCK

(Bild: Shutterstock)

Verifysoft TECHNOLOGY

MIT STATISCHER CODE-ANALYSE ZUM ERFOLG

IoT-Geräte geraten vermehrt in den Fokus von Cyber-Kriminellen. Der klassische IT-Ansatz, Schwachstellen erst im Feld zu beseitigen, kann fatal sein. Besser ist, den Fokus auf die Qualität und Sicherheit des Codes zu legen – bereits zu Beginn der Entwicklung.

Von Klaus Lambertz

Im medizinischen Imaging-Bereich laufen 83 Prozent aller Geräte mit Betriebssystemen, für die es keinen Support gibt. Gar 98 Prozent des Datenverkehrs von IoT-Geräten sind nicht verschlüsselt und 57 Prozent aller IoT-Geräte sind anfällig für schwerwiegende und mittelschwere Angriffe. Das ist das Ergebnis einer Analyse von 1,2 Millionen IoT-Geräten in IT-Abteilungen und Organisationen des Gesundheitswesens in den USA der Forschungsgruppe „Unit 42“ von Palo Alto Networks. Ein Grund für die hohen Prozentsätze: Über die Hälfte der Geräte basiert noch immer auf Windows 7. Für Angreifer ist das ein gedeckter Tisch – und für Embedded-Entwickler herausfordernd. Sie können sich beim Entwickeln neuer Anwendungen und Systeme nicht darauf verlassen, dass die Best Practices bei Fragen der Sicherheit umfassend erfüllt sind. Somit müssen sie selbst ihre Systeme so sicher wie möglich machen.

Viele Angriffsvektoren in der IT und bei Embedded-Systemen machen sich klassische Schwachstellen wie Buffer Overruns oder Null-Pointer-Dereferenzierungen zunutze. Hierbei handelt es sich fast immer um Programmierfehler, die zwar schnell passieren, jedoch nur schwer aufzudecken sind.

Letztlich liegt das an strukturellen Schwächen der hauptsächlich eingesetzten Sprachen C und C++. Beim Entwickeln von Embedded-Software sind C und C++ immer noch die populärsten Programmiersprachen. Jedoch ist die Definition, woraus ein zulässiges C-Programm besteht, der Flexibilität zuliebe sehr liberal ausgelegt – die Compiler können viele Fehler nicht aufdecken. Zudem existieren zahlreiche Mehrdeutigkeiten, die die Compiler, basierend auf unterschiedlichen Interpretationen des Standards, auflösen müssen.

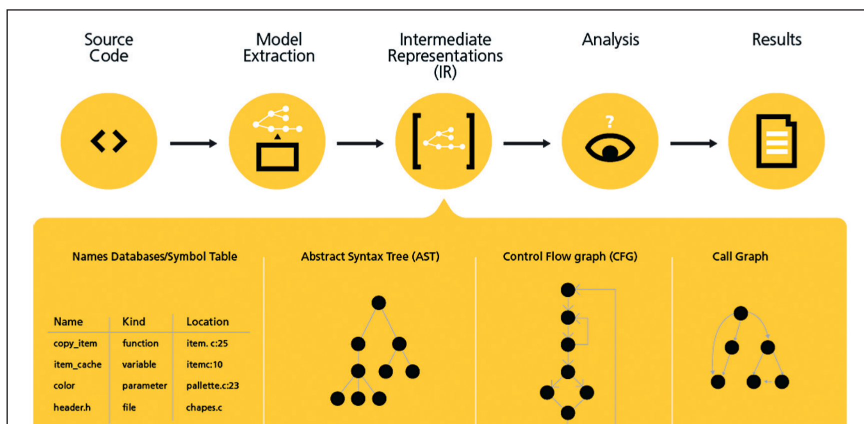

Bild 1. Bei der statischen Code-Analyse wird der Code in eine Intermediate Representation überführt, anhand derer alle Steuerungs- und Datenströme überprüft werden. (Bild: Grammatech)

UPDATES NICHT IMMER MÖGLICH

In der herkömmlichen IT sind es die Anwender gewohnt, dass Fehler fortlaufend mithilfe von Updates behoben werden oder Systeme mit separaten Security-Produkten zu schützen sind. In der Embedded-Welt ist die Situation jedoch eine andere. Zum einen reicht es nicht, zu gegebener Zeit ein Update auszurollen: Ein Buffer Overflow im Herzschrittmacher, unbefugter Fernzugriff auf ein fahrendes Auto oder eine Manipulation von Sensordaten in der Industrie können immense Schäden anrichten. In den letzten Jahren hat sich gezeigt, dass es sich hierbei nicht um Bedrohungen aus einem Science-Fiction-Roman handelt. So gab es etwa Rückrufaktionen bei einer Insulinpumpe und bei einem Herzschrittmacher, da sie Sicherheitslücken aufwiesen, die Angriffe möglich machten. Glücklicherweise wurden die Angriffsvektoren nicht genutzt. Zum anderen sind die Ansätze der traditionellen IT im Embedded-Bereich schlicht nicht praktikabel. Viele Geräte sind mit sehr geringen Übertragungskapazitäten mit dem Internet verbunden. Um die Messdaten eines Sensors zu übertragen, sind wenige Kilobit pro Sekunde völlig aus-

reichend. Für umfassende Software-Updates sind die Kapazitäten jedoch zu gering, zumal in einigen der genutzten Funksegmenten regulatorisch festgelegte Zeitfenster zu berücksichtigen sind. Außerdem sind die meisten Embedded-Geräte weit verteilt oder mobil eingesetzt. Ebenso sind Firewalls zum Schutz vor unerwünschten Aktivitäten lediglich in sehr begrenztem Maße sinnvoll einsetzbar. Aus Kostengründen limitierte Hardware-Ressourcen erschweren Updates zusätzlich.

ERGÄNZUNG ZUM SOFTWARETEST

Ziel muss also sein, die Systeme bereits mit einem Höchstmaß an Sicherheit zu entwickeln und Fehler bereits so früh wie möglich innerhalb des Software Development Lifecycles (SDLC) aufzuspüren. Ein Standardverfahren dazu ist der Softwaretest. Dynamische Tests sind unverzichtbar, um funktionale und strukturelle Schwierigkeiten eines Systems zu erkennen. Sie haben jedoch eine Einschränkung: Ein Test kann Fehler lediglich erkennen, wenn ein Testfall den fehlerhaften Code durchläuft, zu einer Fehlermeldung führt und diese wiederum ein von der Erwartung abweichendes Ergebnis erzeugt. Am Beispiel eines Buffer Overruns wird die Schwierigkeit deutlich. Ein einfacher Buffer Overrun in C könnte so aussehen:

```
char buf[10];
…
buf[10] = ‘a‘ ;
```

Hier wird Speicher für ein Array von zehn Zeichen allokiert, der Zugriff erfolgt auf den elften Index des Arrays. Das Ergebnis des Zugriffs ist nicht definiert. In den meisten Fällen wird das Zeichen „a“ in irgendeinen Speicherbereich neben dem Puffer geschrieben und der vorherige Inhalt dabei überschrieben. Solche Fehler lassen sich recht einfach aufspüren, denn der Index-Wert ist in dem Beispiel hart codiert. In der Praxis jedoch stammen die Indizes meist aus unterschiedlichen Quellen wie Anwendereingaben, Dateien, Signalen von Sensoren und so weiter. Ob etwa ein Buffer Overrun im folgenden Beispiel auftritt, hängt von der Länge des Strings ab, auf den die Variable verweist:

```
int i;
char * s;
s = (char *) malloc(100);
…
i=0;
while (s[i] != ‘\0‘)
i++;
```

Ob sich Fehler wie ein Buffer Overrun beim dynamischen Softwaretest zeigen, hängt von zwei Faktoren ab: Löst der Testfall den Buffer Overrun als Zustandsverletzung aus? Falls ja: Verändert die Zustandsverletzung das erwartete Testergebnis? Der erste Punkt lässt sich mit sehr intensivem Testen erfassen, bei dem die Entwickler ebenso die Auswirkungen von fehlerhaftem Input und ungültigen Werten ausgiebig untersuchen. Bei der zweiten Bedingung wird es schwieriger. Testwerkzeuge sind nicht dazu gedacht, Zustandsverletzungen direkt zu entdecken. Unter

GROSSES RISIKO AUFGRUND KLEINER FEHLER

Oft sind es gerade die kleinen Fehler, die eine Software angreifbar machen und die mithilfe der statischen Analyse relativ einfach zu beseitigen wären. Ein typisches Beispiel dafür ist die Sicherheitslücke in „beep“, einem kleinen Kommandozeilen-Tool bei Linux mit nicht einmal 250 Code-Zeilen. Beep gibt einfach einen Ton auf dem PC-Lautsprecher aus, Frequenz und Dauer sind über Parameter beim Aufruf einstellbar. Bis Version 1.3.4 fand sich hier ein Bug, über den ein Angreifer privilegierte Benutzerrechte auf dem System erlangen konnte. Voraussetzung für einen Exploit des Bugs ist, dass Beep mit „setuid root“ ausgeführt wird. Bei zahlreichen Linux-Distributionen wie etwa Debian war das der Fall, da das Tool einen Schreibzugriff auf die virtuelle Konsole benötigt. Beep parst einige Argumente der Kommandozeile und übergibt den Befehl zum Erzeugen eines Tons an die Systemfunktion „ioctl()“. Über einen Signal-Handler erkennt der Anwender Interrupts. Im Signal-Handler liegt die Ursache für eine „Race Condition“. Eine Folge: Ein Angreifer kann während der Ausführung eine beliebige Datei angeben, in die Beep dann schreibt. So kann die Datei ein symbolischer Link zu jeder beliebigen Datei sein – ebenso zu Systemdateien. Das Data Race betrifft die geteilte Variable „console_device“. Als erster Thread wird das Hauptprogramm erkannt **(Bild a)**, der zweite Thread ist der Signal-Handler **(Bild b)**. Beide greifen auf die Variable console_device zu, ohne dass ein Locking oder ein anderer Mechanismus zur Synchronisation vorhanden ist **(Bild c)**. Mithilfe der statischen Code-Analyse hätte ein Entwickler den Fehler frühzeitig erkennen können. (Bilder: Grammatech)

```
rev_input (/col0/paul/EW/ew.c)
 7   char* rev_input (char * s) {
 8       char buf[LENGTH_OF_INPUT];
 9       int i, j;
10
11       if(strlen(s) > LENGTH_OF_INPUT)
12           exit(-1);  /* input too long */
13
14       i = 0;
         Event 2: i is set to 0.
           • This determines the position accessed during the buffer overrun later.
         hide
15       j = LENGTH_OF_INPUT;
16       while(j >= 0)
17           buf[i++] = s[j--];
             Buffer Overrun
             This code writes past the end of buf.
               • The byte written is at offset i++ from the beginning of buf, whose capacity is 100 bytes.
                   o The offset exceeds the capacity.
                   o i++ evaluates to 100. See related event 5.
               • The overrun occurs in stack memory.
             The issue can occur if the highlighted code executes.
             See related event 5.
             Show: All events | Only primary events
```

Bild 2. Das Analyse-Tool CodeSonar hat einen potenziellen Buffer Overrun im Code erkannt und liefert dem Entwickler genaue Informationen und Hilfen zur Beseitigung. (Bild: Grammatech)

Umständen erkennt das Testteam die einfachsten Buffer Overruns nicht, solange das Überschreiben des Puffers nicht zu einer falschen Ausgabe oder einem Programmabbruch führt.

SCHWACHSTELLEN ERKENNEN

Um solche Fehler bereits deutlich früher im SDLC aufzudecken, bietet sich die statische Code-Analyse an. Vorzugsweise kommen hierfür Werkzeuge zum Einsatz. Sie führen den Code nicht aus, sondern überführen ihn zunächst in ein Modell **(Bild 1)**. Anhand des Modells überprüft das Analyse-Tool alle Steuerungs- und Datenströme und prüft deren Korrektheit anhand so genannter Checker. Beim Erstellen des Modells aus dem Code arbeitet das Werkzeug ähnlich einem Compiler, der Code wird für das Untersuchen in eine Intermediate Representation (IR) überführt.

Da die statische Analyse alle Zustände berücksichtigt, die das Programm theoretisch einnehmen kann, lassen sich potenzielle Fehler mit deutlich höherer Trefferquote aufspüren. Zudem geben die Analyse-Tools den Entwicklern viele hilfreiche Informationen, die beim Beseitigen der Fehler helfen. Standards zur Softwareentwicklung von sicherheitskritischen Anwendungen, etwa ISO 26262 in der Automobilbranche oder DO-178 B/C in der Luftfahrt, schreiben deswegen die statische Analyse entweder explizit vor oder empfehlen sie dringend. Zu den Fehlern, die im Fokus der Analyse stehen, gehören unter anderem die klassischen Einfallstore für Malware:

- ➔ Buffer Overrun/Underrun
- ➔ Command Injection
- ➔ Integer Overflow of Allocation Size
- ➔ SQL Injection
- ➔ Non-constant Format String

Ein entscheidender Vorteil des Ansatzes ist, dass die statische Codeanalyse keinen lauffähigen Code voraussetzt. Hiermit ist sie in allen Phasen des SDLC einsetzbar. Mit der Prüftiefe ist ebenso der Zeit- und Ressourcenaufwand skalierbar: Je nach den gewählten Checkern benötigt die Analyse mehr oder weniger Rechenzeit. Zudem sind Checker bei professionellen Tools individuell definierbar, um den spezifischen Anforderungen des Unternehmens gerecht zu werden. Im Rahmen agiler Entwicklungsansätze wie Continuous Integration/Continuous Deployment (CI/CD), die ebenfalls in der Embedded-Entwicklung immer weiter verbreitet sind, kann das etwa bedeuten: Am Arbeitsplatz der Entwickler findet eine erste Analyse statt, bevor die Integration in die Mainline erfolgt. Um Zeit zu sparen, kann sich die Analyse hier auf sehr typische Fehler oder das Einhalten von Programmierrichtlinien beschränken. Eine weitere Analyseinstanz kann am Build-System arbeiten. Ebenso bietet es sich an, während der normalen Arbeitszeiten ausschließlich einfache Analysen durchzuführen, um die Produktivität der Teams nicht negativ zu beeinflussen. Für tiefe Untersuchungen, die etwas Rechenzeit erfordern, bieten sich zum Beispiel nächtliche Zeitfenster an. Da professionelle Analyse-Tools wie das von Verifysoft Technology vertriebene „CodeSonar" von GrammaTech ein hohes Maß an Automatisierung ermöglichen, ist eine nicht-überwachte Analyse problemlos möglich **(Bild 2)**.

FEHLER FRÜHZEITIG AUFDECKEN

Die statische Code-Analyse als fester Bestandteil des SDLC kann den Testvorgang um ein frühzeitiges Überprüfen des Codes ergänzen. Vor allem klassische Sicherheitslücken – die sich bei komplexen Projekten kaum vermeiden lassen – können Entwickler somit früh erkennen und beheben. Gerade bei Embedded-Systemen ist eine hohe Codequalität unverzichtbar – nicht nur wegen der Schwierigkeiten bei Updates und Patches. In immer mehr Bereichen müssen die Embedded-Systeme nach strengen Standards zertifiziert sein. Hierfür setzen viele Normen den Einsatz der statischen Analyse implizit oder explizit voraus. Ebenso in Bereichen, in denen die Normierung noch eine untergeordnete Rolle spielt, gilt die alte Regel: Je früher ein Fehler gefunden wird, desto weniger Kosten verursacht das Beseitigen. TS

KLAUS LAMBERTZ

ist Geschäftsführer von Verifysoft Technology. Er studierte Betriebswirtschaftslehre an der Fachhochschule Köln und dem Institut Supérieur de Gestion in Paris. Von 1993 bis 1999 verantwortete er den Export bei zwei der größten Druckereien Frankreichs. Vor Gründung von Verifysoft Technology im Jahr 2003 hatte Klaus Lambertz verschiedene Positionen im Vertrieb und im Management mehrerer Unternehmen im Bereich Software-Testing inne.