

SecFlow: Automatische Ermittlung sicherheitskritischer Datenflüsse in Quellcode

Dr. Holger Peine und Stefan Mandel
Fraunhofer Institut für Experimentelles Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern

Dana Richter
CC GmbH
Flachstraße 13
65197 Wiesbaden

Kurzfassung

Je stärker softwarebasierte Systeme in einer vernetzten Welt miteinander gekoppelt werden, desto mehr Angriffspunkte für böswillige Manipulationen entstehen auch, und desto weitreichender werden die möglichen Folgen eines solchen Angriffs für das Gesamtsystem. Die Sicherheit von Software gegen Angriffe rückt damit immer stärker ins Zentrum der Software-Entwicklung. Ein wichtiger Ansatz dabei ist der Einsatz von Werkzeugen zum automatischen Finden von Sicherheitsschwachstellen im Quelltext der Software, denn solche Werkzeuge lassen sich – problemlose Benutzbarkeit vorausgesetzt – einfach in einen Entwicklungsprozess integrieren und auch für bereits existierende (Legacy-) Software nutzen.

Innerhalb des Verbundprojektes SecFlow wird nun ein konfigurierbares, weitgehend programmiersprachenunabhängiges Rahmenwerk konzipiert, das bezüglich einer konfigurierbaren Programmierumgebung nicht vertrauenswürdige Eingabe- und Umgebungsdaten identifiziert, deren programminterne Verarbeitung verfolgt und so ermittelt, ob und wie solche Eingabedaten ungeprüft in sicherheitskritische Operationen der Software einfließen. Eine solche Analyse liefert Schwachstellen, die ganz maßgeblich für die Verwundbarkeit heutiger Software verantwortlich sind und die einem Großteil der bisher bekannt gewordenen Angriffe auf Software zugrunde liegen. Die tief greifende, systematische Analyse aller Datenflüsse der Anwendung auf Sicherheitsrelevanz und fehlende Datenvalidierung wird ein Alleinstellungsmerkmal des zu entwickelnden Werkzeugs sein. Die Sprachunabhängigkeit des angestrebten Frameworks stellt dabei eine Herausforderung dar, ist zugleich aber ein weiteres wesentliches Alleinstellungsmerkmal.

Der Paradigmenwechsel in der IT-Sicherheit

Die IT-Sicherheit stellt aktuell eines der am schnellsten wachsenden Felder der IT-Branche dar. Die Gründe dafür liegen zum einen im immer größeren Gewicht der IT-Systeme als Basis für funktionierende Geschäfts- und Produktionsprozesse – verdeutlicht und noch forciert durch regulatorische Randbedingungen wie KonTraG und Basel-II – und zum anderen in der

stetig zunehmenden Zahl von Angriffen und der gleichsam zunehmenden Verfügbarkeit des nötigen Wissens bzw. der nötigen Werkzeuge für Angriffe auf IT-Systeme.

Technisch steht die IT-Sicherheit gerade vor einem Paradigmenwechsel: Bisher haben sich Angriffe vor allem auf der Ebene der Netzwerkinfrastruktur abgespielt. Die nötigen Sicherheitsmaßnahmen auf dieser Ebene wie z.B. Firewalls werden inzwischen aber immer besser verstanden, so dass sich Angriffe in jüngster Zeit auf die darüber liegende Ebene der Anwendungssoftware konzentrieren. Wissen über die Sicherheit auf Anwendungsebene ist aber bisher nur bei wenigen Experten verbreitet, was Maßnahmen zur Verbesserung der Anwendungssicherheit entsprechend schwierig und kostspielig macht.

Forscht man nach den Ursachen all der IT-Sicherheitsprobleme, die fast täglich gemeldet werden, so stellt man fest, dass neben Fehlern in der Konfiguration von Systemen und Diensten (z.B. für jedermann ohne Identitätsnachweis offen stehende Dienste) und Fehlern in der Benutzung (z.B. vertrauensseliges Klicken auf Links in E-Mails von Unbekannten) ein großer Teil der Schwachstellen letztlich auf Fehler in der Programmierung der eingesetzten Software zurückzuführen ist.

Wie wichtig die saubere Programmierung für die IT-Sicherheit ist, wird erst in jüngster Zeit den Verantwortlichen deutlich. Die Erkenntnis nimmt zu, dass bisherige Sicherheitsmaßnahmen wie z.B. Firewalls oder Virens Scanner nicht die Ursache der Probleme beseitigen (also die Programmierfehler in den Anwendungen), sondern die bestehenden Schwachstellen lediglich überdecken, indem ihre Ausnutzung erschwert wird. Solche "Reparaturmaßnahmen" sind aber in der Praxis nie wirklich vollständig. Hinzu kommt, dass die Verteidiger den Angreifern stets hinterher laufen. Besser ist es deshalb, Software von vornherein sicher zu entwickeln: Auch wenn Maßnahmen wie Firewalls und Virens Scanner wohl niemals überflüssig werden, gilt auch bei der IT-Sicherheit "Fehlervermeidung ist besser als Fehlerbehebung".

SecFlow – IT-Sicherheit bereits während der Programmierung

Böswillige Eingabe- und Umgebungsdaten

Eine zentrale Klasse von Angriffen auf Software beruht auf vom Angreifer geschickt gewählten Eingabedaten, die ohne ausreichende Prüfung als Parameter für sicherheitskritische Operationen der Software benutzt werden. Ein Angreifer tippt zum Beispiel in das Benutzernamenfeld einer Anmeldeseite keinen normalen Namen, sondern einen SQL-Datenbank-Befehl, und die dahinter stehende Anwendung leitet diesen „Namen“ ungeprüft an eine Datenbank weiter, die den Befehl dann „unwissentlich“ ausführt und dadurch Schaden nimmt. Solche ungeprüften Datenflüsse von Eingaben zu kritischen Operationen müssen in der Software gefunden werden, um sie dann durch Einfügen geeigneter Prüfungen entschärfen zu können. Von Hand ist dies fehleranfällig und teuer; Software-Werkzeuge für diesen Zweck gibt es noch nicht in der erforderlichen Analyseschärfe und Konfigurierbarkeit.

Das Verbundprojekt SecFlow widmet sich deshalb der Entwicklung eines solchen flexiblen, konfigurierbaren Werkzeugs zur Ermittlung ungeprüfter, sicherheitskritischer Kontroll- und Datenflüsse in Quelltext. Das Werkzeug wird Daten- und Kontrollflüsse aufdecken, auf denen böswillige Eingabe- und Umgebungsdaten ungeprüft sicherheitskritische Operationen des Zielprogramms beeinflussen können. Solche Datenflüsse sind der maßgebliche Ansatzpunkt für Angriffe auf Software (siehe Abbildung 1).

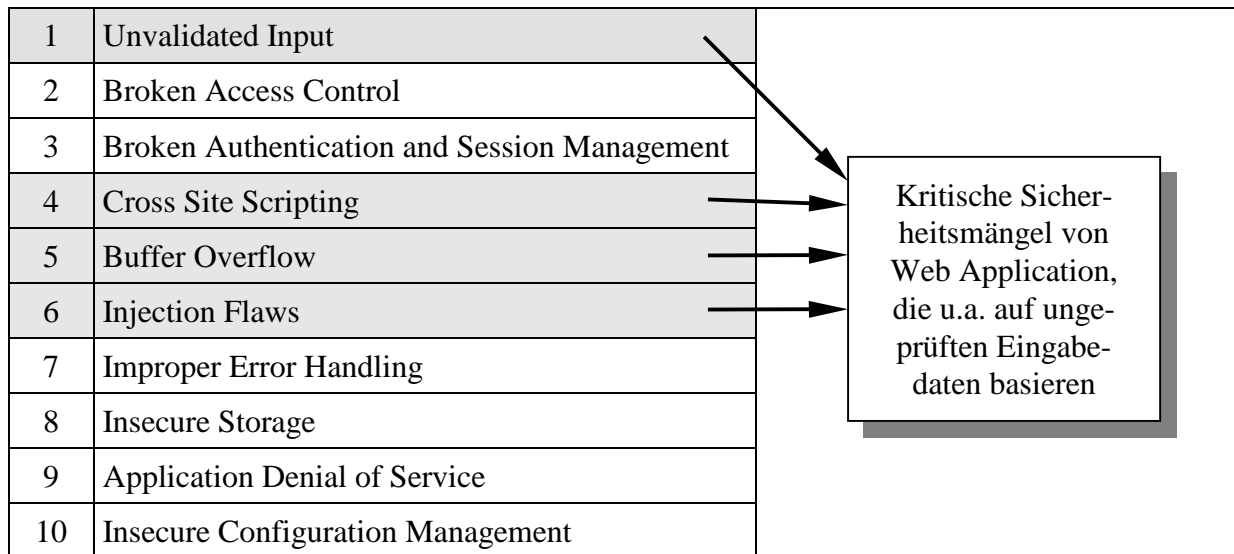


Abbildung 1: Top Ten der kritischen Sicherheitsmängel
 (Quelle: OWASP Open Web Application Security Project: <http://www.owasp.org/>)

Das im Projekt zu entwickelnde Werkzeug soll alle Datenflüsse im betrachteten Programm-Quelltext finden, die von nicht vertrauenswürdigen externen Datenquellen („Quellen“) zu Aufrufen sicherheitskritischer programmexterner Operationen („Senken“) führen, ohne dass die Daten auf ihrem Weg von der Quelle durch das Programm zur Senke validiert, d.h. auf Konformität mit der vom Programm erwarteten Syntax und Semantik geprüft werden.

Quellen, Senken und Validierungen

Eine Quelle aus Sicht des betrachteten Programms ist jede Operation, durch die Daten von außerhalb des Programms in das Programm hinein gelangen – also jede Art von üblichen Eingabeoperationen (z.B. Empfang von Daten aus einer Netzwerkverbindung), aber auch alle Zugriffe auf externe Aufrufparameter, Umgebungsvariable, das Dateisystem etc.

Quellen können Teil der Programmiersprache sein (z.B. die Aufrufparameter der main()-Funktion in C), der Standardbibliothek der Sprache (z.B. die scanf()-Funktion in C) angehören, oder verschiedensten Bibliotheken zum Zugriff auf das Betriebssystem, auf Middleware, oder auf anwendungsspezifische Funktionen entstammen. In einer Web-Anwendung gehören z.B. die Zugriffsfunktionen auf HTTP-Header, Cookies und URL-Parameter zu den Quellen. Die konkreten Ausprägungen von Quellen sind also sehr vielfältig und oft nicht ohne weiteres Hintergrundwissen als solche zu erkennen; allen Quellen ist aber gemeinsam, dass sie beim Zugriff ein bestimmtes Objekt des zugreifenden Programms (z.B. über den Rückgabewert oder einen Ausgabeparameter einer Funktion) mit externen Daten füllen. Solche Daten gelten grundsätzlich als zunächst nicht vertrauenswürdig: Falls bestimmte Quellen in der konkreten Anwendung als a priori vertrauenswürdig gelten dürfen, so ist dies dem Werkzeug in seinen Konfigurationsdaten anzuzeigen.

Eine Senke aus Sicht des betrachteten Programms ist jede Operation, bei der Daten aus dem Programm einen sicherheitskritischen Ablauf außerhalb des Programms beeinflussen. Typischerweise sind dies Operationen deren Daten als Eingabeparameter bestimmter Systemfunktionen, z.B. von Betriebssystem-Kommandozeilenaufrufen, Datenbankoperationen, oder zu Manipulationen externer Prozesse dienen. Wenn solche kritischen Funktionen mit böswillig

gewählten Daten aufgerufen werden, kann das Programm „ungewollt“ seine eigenen Sicherheitsziele verletzen, z.B. vertrauliche Daten aus einer externen Datenbank offen legen. Analog zu den Quellen gilt auch für die Senken, dass es je nach betrachteter Programmierumgebung nicht ohne weiteres zu erkennen ist, was eine sicherheitskritische Operation ist. Allen Senken ist aber gemeinsam, dass sie beim Zugriff die Daten aus einem bestimmten Objekt des Programms (typischerweise ein Aufrufparameter einer Funktion) zur Beeinflussung der sicherheitskritischen programmexternen Funktion verwenden.

Die Validierung von Daten stellt sicher, dass die zunächst nicht vertrauenswürdigen Daten aus den externen Quellen auf ihre Zulässigkeit überprüft werden, bevor sie – in der Regel indirekt über eine verkettete Beeinflussung beliebig vieler anderer Datenobjekte – in eine Senke einfließen dürfen. So wird verhütet, dass fehlerhafte oder böswillig manipulierte Quellen dort unerwünschte, sicherheitsrelevante Effekte hervorrufen. Die konkrete Form der Datenvalidierung hängt vom Datentyp und vom Verwendungszweck der Daten ab – ein Integer-Wert könnte z.B. einfach mit einer unteren und oberen Schranke verglichen werden, während ein String-Wert vielleicht anhand der Übereinstimmung mit einem regulären Ausdruck validiert wird. Im Gegensatz zu den Quellen und Senken ist es allerdings nicht realistisch, alle Validierungen automatisch zu erkennen; z.B. könnten syntaktisch validierungsähnliche Operationen tatsächlich einen ganz anderen Effekt haben.

Validitäts- und Kritikalitätsanalyse von programminternen Daten

Das Werkzeug soll eine Analyse sämtlicher für die Verarbeitung von Eingabedaten relevanter Kontroll- und Datenflüsse des Zielprogramms durchführen. „Sämtlich“ ist dabei im Sinne einer statischen Analyse zu verstehen, d.h. es handelt sich immer um eine bestimmte, endliche Zahl von Daten- und Kontrollflüssen, die sich aus der Struktur des Quelltexts ergibt. Ziel der Analyse ist es, die Validität und die Kritikalität aller Datenobjekte des Programms zu verfolgen. Ein Datenobjekt ist valide an einem bestimmten Punkt in der Programmausführung, falls es entweder aus einer vertrauenswürdigen Quelle stammt (konstante Daten sind z.B. immer valide) oder irgendwann vor diesem Punkt im oben beschriebenen Sinne validiert wurde. Daten, die – direkt oder indirekt – aus einer nicht vertrauenswürdigen Quelle stammen, sind damit invalid, und auch das Ergebnis der Verknüpfung eines validen Operanden mit einem invaliden Operanden ist selbst wieder invalid. Die Invalidität von Daten pflanzt sich also „vorwärts“ durch das Programm fort, bis die Daten validiert werden. Dual zur Validität ist das Konzept der Kritikalität: Ein Datenobjekt ist kritisch an einem bestimmten Punkt in der Programmausführung, wenn es auf wenigstens einem möglichen Kontrollfluss von diesem Punkt aus zu einer Senke im oben beschriebenen Sinn führt. Die Kritikalität pflanzt sich „rückwärts“ fort: Alle Operanden eines kritischen Ausdrucks sind auch selbst kritisch. Zur Implementierung der Validitäts- und Kritikalitätsanalyse soll auf Techniken der Programmanalyse wie Datenflussanalyse, kontextsensitive Pointer-Analyse und Program Slicing aufgebaut werden.

Das Werkzeug wird einen Graphen der statisch möglichen Kontrollflüsse aufbauen und die Validität aller referenzierten Datenobjekte des Programms vorwärts entlang des Kontrollflusses von den Quellen aus über die verschiedenen programmiersprachlichen Operationen hinweg verfolgen. Mögliche Validierungen der Daten werden heuristisch erkannt; dabei müssen auch Zugriffe auf dasselbe Objekt unter verschiedenen „Namen“ (Objektreferenzen) korrekt erfasst werden. Bei der Konstruktion des Kontrollflussgraphen müssen auch schwierige Verzweigungen durch polymorphe Methodenaufrufe oder Exceptions (je nach Zielsprache) berücksichtigt werden. Dual zur Validitätsverfolgung wird das Werkzeug die Kritikalität von Daten durch Rückwärtsanalyse des Kontrollflusses von den Senken aus verfolgen; eine optimal effiziente Verzahnung von Vorwärts- und Rückwärtsanalyse stellt dabei ein herausfor-

derndes Nebenthema dar. Falls bei der Analyse an einem Programmausführungspunkt ein Datenobjekt als sowohl invalid als auch kritisch identifiziert wird, stellt der Kontrollfluss durch diesen Punkt und der Datenfluss durch dieses Objekt einen der gesuchten ungeprüften, sicherheitskritischen Kontroll- bzw. Datenflüsse dar.

Aufrufe programminterner Funktionen werden (bei der ersten Analyse dieser Funktion) durch eine rekursive Analyse des Funktionskörpers behandelt; Aufrufe externer Funktionen werden bei der Validitäts- und Kritikalitätsverfolgung durch ihren validitäts- bzw. kritikalitätsrelevanten Effekt ersetzt (also nicht etwa durch ihren funktionalen Effekt, der eine Ausführung der Funktion erfordern würde). Dieser Effekt muss für jede externe Funktion (bzw. Methode etc.) einmalig vorab ermittelt werden.

Methoden zur Aufdeckung von Sicherheitsschwachstellen

Die Problematik von Sicherheitsschwachstellen ist nicht so einfach wie die von einfachen Programmierfehlern. Letztere können durch funktionales Testen anhand der Anforderungsbeschreibung oder sogar schon zur Übersetzungszeit gefunden werden (moderne Compiler erkennen inzwischen sehr viele Probleme bereits vor dem ersten Lauf). Sicherheitsschwachstellen hingegen werden im ordnungsgemäßen Betrieb gar nicht sichtbar; sie können allerdings durch von arglistigen Angreifern geschickt gewählte Eingaben ausgenutzt werden. Dabei hat jede Sicherheitsschwachstelle ihre individuellen Eigenheiten; es verlangt viel Expertenwissen und Intuition um eine Sicherheitsschwachstelle ohne Wissen über den Quelltext zu finden. Selbst mit Kenntnis des Quelltextes ist eine manuelle Sicherheitsanalyse keineswegs trivial.

Manuelle Methoden zur Aufdeckung von Sicherheitsschwachstellen sind Inspektionen und Tests, wobei Tests im Allgemeinen nur Funktionalität abprüfen können, nicht jedoch fehlende Funktionalität bzw. fehlende Sicherheitsfeatures. Zur Erreichung einer höheren Zuverlässigkeit und Produktivität ist eine teilweise Automatisierung von Methoden innerhalb des Software-Prozesses unerlässlich.

Auf Grund der Komplexität ist die automatisierte Entdeckung von Sicherheitsschwachstellen nicht einfach zu bewerkstelligen.

Die Idee, Sicherheitsprobleme erst zur Laufzeit zu überprüfen, da hier die Eingaben und Ausgaben bereits bekannt sind, ist eine Möglichkeit, das Problem mit der Eingabe/Ausgabe-Abhängigkeit von Schwachstellen zu bewältigen. Dieser Schritt lässt sich jedoch nur in interpretierten Sprachen vollziehen, da nur bei diesen noch korrektive Maßnahmen während der Laufzeit möglich sind. Ein bekanntes Beispiel hierfür ist der so genannte Taint-Modus im Perl-Interpreter, der während der Laufzeit den Sicherheitsstatus von Variablen überwacht. Der Taint-Modus ist eigentlich eine sehr primitive Maßnahme, die theoretisch viel spezifischer optimiert werden könnte. Es ist anzunehmen, dass solche Anstrengungen nicht unternommen wurden, da jede weitergehende Analyse auch ihren Teil von der Performanz des Interpreters fordern würde.

Eine andere Möglichkeit für Neuentwicklungen ist die Erzeugung von sicherem Code. Relevant ist diese Technik vor allem für maschinennahe Programmiersprachen wie C, welche keine speziellen Vorkehrungen bei der Übersetzung treffen. Für Programme in C stehen bereits Module bereit, die zumindest Speicher sicher belegen und freigeben können (verhindert Buffer-Overflow-Schwachstellen). Da diese Bibliotheken nicht automatisch eingebunden und verwendet werden, ist die Sicherheit hier natürlich weiterhin abhängig von der Konsequenz des Entwicklers. Außerdem helfen solche Bibliotheken nur, die fundamentalen Probleme mit der Ressourcenverwaltung zu beheben und eine maschinennahe Sprache damit auf den Sicherheitslevel einer modernen, höheren Programmiersprache (wie z.B. Java oder Smalltalk)

zu heben. Eine Sanierung von Sicherheitsschwachstellen in größeren Altsystemen ist mit dieser Technik nicht möglich.

Methoden der Quellcode-Analyse

Eine Möglichkeit, die für alle Programmiersprachen zur Verfügung steht und die auch für Altsysteme noch anwendbar ist, ist die Quellcode-Analyse. Die Forschung an entsprechenden Werkzeugen hat in den vergangenen Jahren Fortschritte gemacht – von syntaxgesteuerten Werkzeugen, über strukturgesteuerte Werkzeuge, flussgesteuerte Werkzeugen zu Werkzeugen, die die Laufzeitsemantik prüfen (Modelchecking-Werkzeuge). Mit fortschreitender Komplexität nimmt die Qualität der Ergebnisse zu. Die Performanz jedoch sinkt und der zusätzliche Aufwand für Quellcodevorbereitung steigt. Die am weitesten verbreiteten freien Programme zur Quellcode-Analyse sind syntaxgesteuert oder strukturgesteuert. Die kommerziellen Werkzeuge sind meist strukturgesteuert oder flussgesteuert. Rein modelchecking-basierte Werkzeuge sind derzeit noch in der experimentellen Phase.

Syntaxgesteuerte Werkzeuge verwenden als Basis für ihre Analysen eine Repräsentation, die sehr nahe am Quelltext liegen kann. Oft reicht eine lexikalische Analyse aus, die Wortketten ausgibt; einige Ansätze verwenden auch einen Syntaxbaum, der durch einen Parser generiert wird. Eine Auflösung von Variablen und Typen findet nicht statt. Die Analysen bestehen meist nur auf einer Analyse des Zusammenhangs zwischen Schlüsselworten und Schlüsselementen. Die Suche erfolgt in der Regel über reguläre Ausdrücke auf Wortketten. Gesucht werden spezifische Bibliotheksfunktionen (mit bekannten Schwachstellen) und dazu verdächtige Kontexte. Manchmal müssen diese Analysen noch durch manuelle Optimierungen unterstützt werden, da die Ausgabe ohne diese Optimierungen zu schlechter Qualität vorweisen würden. Diese Optimierungen sind spezifisch für das konkrete Problem und normalerweise in das Werkzeug fest eingebaut, so dass sich keine Verbesserungen für andere Schwachstellen ergeben. Trotz dieser Optimierungen ist die Anzahl falsch positiver Befunde vergleichsweise hoch. Darüber hinaus ist die Schwachstellen-Abdeckung von syntaxgesteuerten Techniken sehr spezifisch.

Strukturgesteuerte Werkzeuge verwenden üblicherweise auch einen Syntaxbaum für die Analyse. Eine Typ- und Namensanalyse ist hier möglich. Die Analyse erfolgt unter Berücksichtigung der Strukturen im Baum. Durch die Möglichkeit, auch die Herkunft von Variablen und Prozeduren bestimmen zu können, kann auch der nicht lokale Kontext in die Analyse mit einbezogen werden. Weiterhin möglich ist die Einbeziehung von Metriken in die Analyse, um aufgefundene Schwachstellen zu bewerten.

Gesucht werden kann hier auch nach Bibliotheksfunktionen und verdächtigen Kontexten, wobei der Kontext hier auch als strukturelles Muster spezifiziert werden kann. Teilweise werden auch hier reguläre Ausdrücke eingesetzt, allerdings operieren diese nicht mehr auf einer Wortkette, sondern auf einem Baum und haben dadurch wesentlich komplexere Erkennungsmöglichkeiten. Insgesamt sind auch die strukturgesteuerten Werkzeuge sehr spezifisch, aber durch das komplexere interne Modell sind die Ergebnisse oft genauer als die der syntaxgesteuerten Werkzeuge. Komplexere strukturgesteuerte Werkzeuge werden auch kommerziell vertrieben, was für eine annehmbare Leistungsfähigkeit dieses Ansatzes spricht. Diese Werkzeuge verpacken oft mehrere spezifische Analysen zu einem großen Werkzeug zusammen.

Die Tatsache, dass syntaxgesteuerte und strukturgesteuerte Ansätze sich auch weiterhin behaupten, liegt darin begründet, dass sie wesentlich einfacher zu bedienen sind und einfach definierte Prüfungen durchführen.

Flussgesteuerte Programmanalysen analysieren nicht nur die statische Struktur eines Quellcodes, sondern auch die Daten- und Kontrollflüsse, die hier auftreten. Da das konkrete Wissen

Wissen über Eingabedaten nicht vorhanden ist, ist es gängige Praxis, konservative Analysen durchzuführen, die immer das schlimmstmögliche Verhalten unterstellen und bei der Analyse ausgeben.

Der Vorteil dieses Verfahrens gegenüber den syntaxgesteuerten und strukturgesteuerten ist, dass hier Datenfluss- und Kontrollfluss-Muster gesucht und bewertet werden können. Durch den Datenflussgraphen wird implizit eine Abhängigkeitsrelation der Variablen untereinander gegeben, und durch den Kontrollflussgraphen eine Abhängigkeitsrelation der Prozeduraufrufe untereinander. Die Suche erfolgt durch eine statische und dynamische Spezifikation des zu suchenden Musters. Es ist so möglich, nicht nur konkrete Variablen, sondern auch die Abhängigkeiten zu anderen relevanten Variablen in die Suche einzubeziehen. Genauso ist es möglich, Prozedurprotokolle (welche Prozeduren vor welchen ausgeführt werden müssen, welche sich nicht gegenseitig aufrufen dürfen, etc.) in eine Suchanfrage mit einzubeziehen. Insbesondere erlaubt die Suche, potentiell gefährliche Kontexte zu finden ohne konkretes Wissen über die zentrale anfällige Funktion zu kennen. Als Ergebnis könnte eine solche Suche auch potentiell gefährdete Funktionen ausgeben. Das unterscheidet flussgesteuerte Analysen deutlich von den primitiveren Analysen, denn hier geht die Suche immer von der anfälligen Funktion aus, unbekannte Funktionen werden nie als Schwachstelle erkannt.

Durch die Möglichkeit, Daten- und Kontrollflüsse über die gesamte analysierte Codebasis zu verfolgen, kann diese Form der Analyse auch selbstständig ermitteln ob sich im analysierten Quellcode unsichere Prozeduren befinden. In verschiedenen Situationen mag es durchaus sinnvoll sein, eine Basisfunktionalität nicht sicher zu gestalten und die Validierung nur dann durchzuführen, wenn der Wert auch tatsächlich aus einer vertrauensunwürdigen Quelle kommt oder zu einer Senke fließt, die eine Validierung zwingend erfordert.

Strikt konservative Analysen erzeugen noch viele falschpositive Ergebnisse. Deswegen ist es anzunehmen, dass diverse Analysewerkzeuge von der strikt konservativen Analyse abgegangen sind und stattdessen eher die Performanz und die Qualität des Ergebnisses anstreben. Neuere Forschungen haben versucht, diesem Phänomen mit Bewertungsalgorithmen zu begegnen, die die ausgegebenen Ergebnisse nach ihrer Relevanz ordnen. Andere verifizieren die Ergebnisse der statischen Analyse mit einer nachgeschalteten dynamischen Analyse oder einem Modelchecking.

Der fortgeschrittenste Ansatz zur Analyse von Programmen auf Quell-Code-Ebene ist die modellchecking-basierte Analyse. Modelchecking wird normalerweise zur Verifikation von Programmen eingesetzt, nicht zur Analyse. Es handelt sich um eine Variante der formalen Programmverifikation, bei der nur ein Teil des Systems modelliert wird (deswegen Modell) und der Teil mit dem unwichtigen Verhalten nicht ins Modell eingeht. Der erste Schritt des Modelchecking ist die Spezifikation eines Problems, das gelöst werden soll. Manchmal lässt sich daraus (und aus dem zu überprüfenden Programm) auch bereits das Modell ableiten. Je nach Komplexität des Problems müssen jedoch für das Modell auch zusätzliche Abstraktionsregeln eingeführt werden. Das Modell selbst kann in verschiedenen Formen repräsentiert werden. Üblich ist die Darstellung als endlicher Automat. In jedem seiner Zustände werden Zusicherungen geprüft, die durch die Spezifikation bzw. das geprüfte Programm gegeben sind.

Wie in der flussgesteuerten Analyse ist es auch bei der modellchecking-basierten Analyse möglich, potentiell gefährliche Kontexte zu finden und somit den Benutzer auch auf gefährdete Funktionen hinzuweisen.

Der Vorteil des Modelchecking gegenüber den vorangehenden Techniken ist, dass man hiermit dynamische Eigenschaften zuverlässig spezifizieren, verifizieren und widerlegen kann. Von der Theorie her ergibt dieses Verfahren keine falschpositive Ergebnisse, sofern die Spezifikation entsprechend scharf formuliert wurde. Allerdings erfordert es Erfahrung mit Model-

checking, um eine geeignete Spezifikation zu erstellen. In der Praxis ist die Erstellung einer Spezifikation in der gewünschten Genauigkeit nicht wirtschaftlich. Aus Studien zu dem Thema geht zudem hervor, dass sich Modelchecking zur Spezifikation von strukturellen (nicht dynamischen) Schwachstellen nur sehr bedingt eignet. Modelchecking ist auch ohne die vorausgehenden Schritte der Spezifikation sehr langsam. Diese Nachteile könnten ein Grund dafür sein, dass Modelchecking oft nur in Kombination mit flussgesteuerten Techniken eingesetzt wird.

Ein Nachteil der modelchecking-basierten Analyse ist, dass die Problemspezifikationen nicht einfach generisch formuliert werden können, d.h. es ist recht einfach, ein Programm auf Sicherheit an einer konkreten Stelle zu prüfen, aber schwierig, es auf Sicherheit an allen möglichen Stellen zu prüfen.

Grenzen der Methoden

Allen Ansätzen gemein ist die Problematik, dass falschpositive Ergebnisse die Qualität der Analyse senken. Einige Werkzeuge versuchen diesem Problem zu begegnen, indem der Anwender vor der Benutzung den Quelltext an allen „interessanten“ Stellen mit Steueranweisungen (z.B. in Form von Annotationen) für das Werkzeug instrumentieren muss. Im Fall von syntaxgesteuerten oder strukturgesteuerten Analysen können so die relevanten Kontexte ausgezeichnet werden. In der flussgesteuerten Analyse kann man die kritischen Ein-/Ausgabefunktionen so kennzeichnen und auch die Semantik der Validierungen. Beim Modelchecking können Annotationen Hilfen für die Generierung des Modells bilden. Insgesamt liefern Annotationen zwar bessere Ergebnisse, sie machen aber die Benutzung des Werkzeugs für größere Quelltexte oder vorhandene Software-Altbestände zu einem sehr aufwändigen und lästigen Unterfangen.

Schließlich sind alle bisherigen Werkzeuge nur für Sicherheitsexperten benutzbar, was ihre Benutzung weiter verteuert und ihre Integration in den üblichen Arbeitsablauf eines Entwicklers und in den gesamten Entwicklungsprozess (z.B. einschließlich täglicher, automatischer Tests) erschwert. Keines der bisherigen Prüfwerkzeuge lässt sich so genau auf bestimmte Eigenschaften der Zielsoftware einstellen, dass es auch von Sicherheitslaien oder gar vollkommen automatisiert eingesetzt werden kann.

Werkzeugarchitektur und zielsprachenspezifische Aspekte

Die zentralen Funktionen des SecFlow-Werkzeugs, vor allem die Validitäts- und Kritikalitätsanalyse der Daten- und Kontrollflüsse, werden in einem zielsprachenunabhängigen Kern zusammengefasst. Dieser Kern wird eine klar definierte Schnittstelle für sprachspezifische Module beinhalten. Während des Projektes werden Sprachmodule für die Zielprachen Java und C# sowie für die Zielplattformen J2EE und .NET entwickelt werden. Der Entwurf der Schnittstelle wird aber bereits einer möglichen Anbindung weiterer Sprach-Plugins z.B. für Programmiersprachen wie C, Visual Basic oder PHP Rechnung tragen. Ein Sprachmodul muss dann dem Kern unter anderem folgende sprachspezifische Funktionalitäten und Daten an der Schnittstelle zur Verfügung stellen können:

- eine Liste sprachspezifischer, gegebenenfalls auch bibliotheks- oder anwendungsspezifischer Datenquellen, Datensenzen und Validierungsheuristiken (zu formulieren in einem sprachunabhängigen Beschreibungsformalismus)
- eine Repräsentation des Zielprogramms in dem vom Werkzeug erwarteten Format, sowie weitere sprachspezifische Komponenten wie Symboltabellen etc.

Bei der Implementierung der Parser und ähnlicher Standard-Komponenten soll auf verfügbare Toolkits zurückgegriffen werden.

Das Werkzeug wird eine Vielzahl von Heuristiken einsetzen, die typische Code-Muster für bestimmte Sicherheitsschwachstellen oder bestimmte Validierungen erkennen können. Dabei werden neben allgemeingültigen auch plattform- und anwendungsspezifische Heuristiken beschrieben werden. Zum Beispiel ist das Code-Muster `if (x > MAX) { return OUT_OF_RANGE; }` eine Heuristik dafür, dass hier die Variable `x` validiert wurde. Durch anwendungsspezifische Konfigurationshinweise kann die Genauigkeit der Heuristiken noch weiter geschärft werden (z.B. „in dieser Anwendung ist `OUT_OF_RANGE` ein Fehlercode, der ungültige Parameter anzeigt“), so dass die Hoffnung besteht, einen Großteil der Validierungen erkennen zu können. Das Werkzeug wird aber auch bereits mit der anwendungsunabhängigen Default-Konfiguration brauchbare Ergebnisse liefern. Damit wird es auch für Anwender mit nur oberflächlichen Security-Wissen erfolgreich einsetzbar sein.

Die grundlegenden Konzepte und Techniken des entwickelten Werkzeug-Frameworks und wesentliche Anteile der entwickelten Software bereits während des Projektes der Öffentlichkeit zur Verfügung zu stellen.

Zusammenfassung

Ein Großteil der Sicherheitsschwachstellen beruhen auf Programmierfehlern, die auf fehlendes Wissen der Entwickler über Software-Sicherheit zurückzuführen sind. Zudem treten viele dieser Fehler immer wieder auf, besonders Versäumnisse bei der Eingabedatenvalidierung. Es erscheint daher grundsätzlich sinnvoll, nach solchen Schwachstellen automatisch zu suchen.

Menschliche Experten nutzen bei einer Sicherheitsanalyse von Quelltexten ihr umfangreiches Wissen über die möglichen Eingabekanäle der jeweiligen Programmierplattform, sind aber durch die für Menschen mühselige Verfolgung der internen Programmabläufe nicht besonders effizient bei solchen Analysen. Es erscheint daher viel versprechend, das Expertenwissen in einem Werkzeug zu kodifizieren, das auch die mühsame und fehlerträchtige Arbeit der programminternen Ablaufverfolgung übernehmen kann.

Da die möglichen Eingabepunkte und potentiellen Schadstellen im Quelltext grundsätzlich erkennbar sind, und auch die Validierung typischer Daten in den meisten Fällen (eventuell mit Hilfe von vorkonfiguriertem Wissen über anwendungsspezifische Validierungsmuster) durch ein Analysewerkzeug erkannt werden kann, erscheint das geplante Werkzeug technisch zwar ehrgeizig, aber grundsätzlich machbar und im Erfolgsfall überaus nützlich.

Das größte Risiko des Projekts liegt in der Erkennung der vielfältigen Validierungsoperationen für Objekte. Korrekt validierte Objekte, die aber nicht als solche erkannt werden, können zu falsch positiven Befunden von angeblich unsicheren Datenflüssen durch diese Objekte führen. Da, wie bei den Entwurfszielen beschrieben, die weitgehende Vermeidung falsch positiver Befunde ein zentrales Entwurfsziel und Gütekriterium des Werkzeugs ist, muss eine weitreichende Abdeckung der Erkennung von Validierungsoperationen erreicht werden. Da die möglichen Ausprägungen solcher Operationen sehr vielfältig sind, wird eine Erkennung nur auf der Basis einer Menge von Heuristiken möglich sein. Die bisher geplanten Heuristiken für die Validierung von Standard-Datentypen geben aber Anlass zum Optimismus, dass schon eine rein heuristische Validierungserkennung die Mehrheit der Validierungsoperationen identifizieren kann. Der nicht heuristisch erkennbare Anteil wird vor allem aus anwendungsspezifischen Validierungen bestehen; genau solche Eigenheiten des konkreten Quelltexts sollten

sich aber gut durch eine anwendungsspezifische Konfiguration des Werkzeugs erfassen lassen.

Der Einsatz eines solchen Werkzeugs ist in einem Unternehmen mit stark unterschiedlich verteilter Sicherheitskompetenz auf Seiten der Entwickler vor allem dann sinnvoll und wirtschaftlich, wenn es nicht nur für Sicherheits-Experten, sondern auch für Entwickler gut zu benutzen ist, die relative Sicherheits-Laien sind. Diesem Aspekt wird dadurch Rechnung getragen, dass das Werkzeug von Experten so zielgenau konfiguriert werden kann, dass sein Einsatz, auch dank der für Software-Entwickler intuitiven Befunddarstellung, keinen größeren Aufwand mehr verursacht.