Iterative präzisionsbewertende Signaturgenerierung

René Rietz, Sebastian Schmerl, Michael Vogel und Hartmut König

Brandenburgische Technische Universität
Lehrstuhl Rechnernetze und Kommunikationssysteme
03013 Cottbus, Postfach 10 13 44
{rrietz, sbs, mv, koenig}@informatik.tu-cottbus.de

Abstract: Die Wirksamkeit signaturbasierter Intrusion Detection Systeme hängt entscheidend von der Präzision der verwendeten Signaturen ab. Die Ursachen unpräziser Signaturen sind hauptsächlich der Signaturableitung zuzuschreiben. Die Spezifikation einer Signatur ist aufwendig und fehleranfällig. Methoden für ein systematisches Vorgehen existieren bisher kaum. In diesem Papier stellen wir einen Ansatz zur systematischen Ableitung von Signaturen für Host-basierte IDS vor. Ausgehend vom Programmcode und der Verwundbarkeit werden ganze Signaturen oder Signaturfragmente generiert. Wir zeigen, dass durch den Einsatz von statischer Code-Analyse der Entwurfsprozess für Signaturen automatisiert und entscheidend verkürzt werden kann. Ferner ist eine Qualitätsabschätzung der abgeleiteten Signatur möglich.

Keywords: Intrusion Detection, Signaturanalyse, Signature Engineering, systematische Signaturableitung, Quellcodeanalyse

1 Einleitung

Um den Gefahren eines sich ständig vergrößernden Bedrohungspotentials für IT-Infrastrukturen zu begegnen, werden zunehmend Intrusion Detection Systeme (*IDS*) eingesetzt. In der praktischen Anwendung erweist sich die Signaturanalyse im Vergleich zur Anomalieerkennung bisher als die effizientere und zuverlässigere Variante. Sie steht hier im Mittelpunkt der Betrachtung. Signaturbasierte Analyseverfahren untersuchen Protokoll- oder Auditdaten nach Mustern bekannter Sicherheitsverletzungen, den *Signaturen*. Die Wirksamkeit der Analyse hängt entscheidend von der Präzision der verwendeten Signaturen ab. Unpräzise Signaturen schränken die Erkennungsfähigkeit der Analyse stark ein und führen zu den typischen hohen Fehlalarmraten. Die Ursachen dafür sind hauptsächlich in der manuellen Ableitung von Signaturen aus vorliegenden Angriffsszenarien zu finden. Diese Ableitung erfolgt meist empirisch auf der Grundlage des Wissens und der Erfahrung von Experten.

Die in der Forschung untersuchten Ansätze zur Automatisierung der Signaturableitung lassen sich in zwei Klassen gliedern: (1) Black-Box-basierte Verfahren, die auf der Analyse von Angriffsprogrammen basieren und (2) White-Box-basierte Verfahren, die den

Quelltext bzw. Binärcode der verwundbaren Anwendung untersuchen. Bei den Black-Box-basierten Verfahren werden häufig die Angriffsspuren mehrerer realer Angriffe (Netzwerkpakete) auf gemeinsame Substrings untersucht und mittels Graphclusterungsalgorithemen zu Signaturen zusammengefasst. Beispiele hierfür sind POLYGRAPH [NKS05] und NEBULA [WFGPM09]. Die White-Box-basierten Verfahren können weiter unterteilt werden in: (2.1) Verfahren, die aus den Angriffsspuren (hier: *execution traces*) und einem Programm- oder Datenmodell die für eine Verwundbarkeitsausnutzung notwendigen Eingaben berechnen [BNS+06, CCZ+07, CPWL07] sowie (2.2) Policy-basierte Verfahren [FHSL96, WD01, SBD01, GJM04], die das Programmmodell selbst als Signatur auffassen und vom Modell abweichendes Programmverhalten als Angriff interpretieren. Die Verfahren (1) und (2.1) benötigen mindestens ein Angriffsprogramm. Der Signaturgenerierungsprozess beginnt demzufolge erst nach der Verbreitung von Angriffsprogrammen und die generierten Signaturen müssen infolgedessen schneller als die Angriffsprogramme verbreitet werden. Die Verfahren aus (2.2) erfordern kontinuierliche Vergleiche zwischen dem Programmverhalten und dem zugehörigen Programmmodell. Für die Gewährleistung einer geringen Fehlalarmrate bei der Ermittlung von abweichendem Programmverhalten sind sehr genaue Programmmodelle erforderlich, die wegen ihres Umfangs (Ressourcenbedarf) in der Praxis nicht eingesetzt werden können. Eine Qualitätsabschätzung der generierten Signaturen wird in keinem der genannten Verfahren realisiert.

In diesem Beitrag wird eine Methodik zur Automatisierung der Signaturableitung vorgestellt, die keine Angriffsprogramme erfordert und die eine Qualitätsabschätzung der generierten Signaturen ermöglicht. Das vorgestellte Verfahren generiert zu diesem Zweck ein Programmmodell, das ebenfalls für IDS der Klasse 2.2 verwendet werden kann. Voraussetzung für den hier vorgestellten Ansatz ist die Kenntnis über die Programmstellen, an denen eine Verwundbarkeit erfolgreich ausgenutzt werden kann (*Punkte der Verwundbarkeitsausnutzung*). Diese Programmstellen sind der Ausgangspunkt für die Generierung einer Signatur zur Verwendung in Host-basierten Intrusion Detection Systemen. Ziel ist hierbei, dass durch die generierte Signatur erkannt wird, ob das verwundbare Programm durch Eingaben oder Aktionen (z.B. durch Nachrichten oder Kommandos) in einen verwundbaren Programmzustand überführt wird. Das hier vorgestellte Verfahren ist unabhängig von der konkret verwendeten Signaturbeschreibungssprache.

Wir stellen zunächst das allgemeine Konzept in Form der einzelnen Generierungsschritte vor. Anschließend wird eine prototypische Implementierung des Generierungsprozesses anhand von realen Verwundbarkeiten evaluiert. Der Beitrag schließt mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf die zukünftigen Arbeitsschritte.

2 Generierung von Signaturen mittels Quellcodeanalyse

Der bisherige manuelle Entwicklungsprozess einer Signatur für einen neuen Angriff kann grob in vier Schritte unterteilt werden. (1) Zuerst wird der Angriff ausgeführt, um die entstehenden Spuren (*Auditereignisse*) aufzuzeichnen. Spuren sind einzelne sicherheitsrelevante Aktionen, die durch den Sensor eines IDS protokolliert werden. (2) Im Anschluss untersucht der Signaturmodellierer diese Spuren und identifiziert die zum Angriff

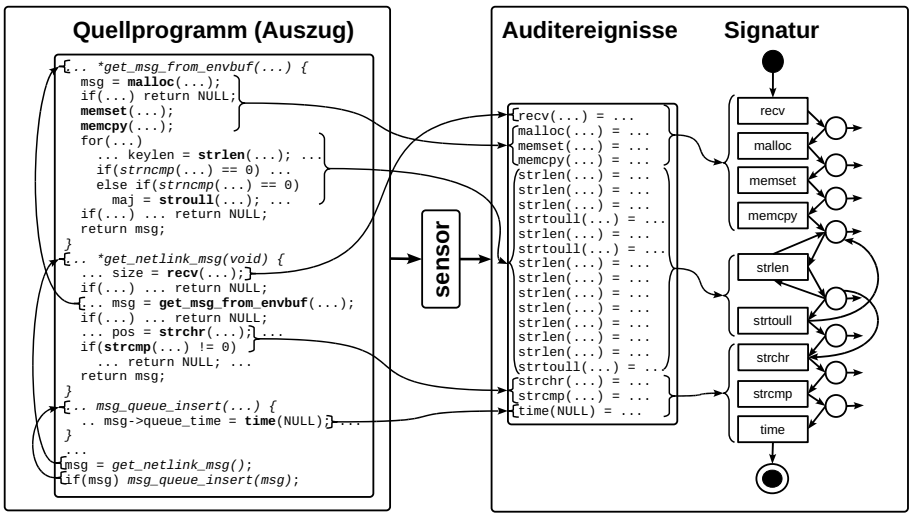


Abbildung 1: Sichtbares Programmverhalten

gehörenden *sicherheitskritischen Aktionen*. Darauf aufbauend entwickelt er (3) schrittweise die neue Signatur. Anschließend wird (4) in einer Testphase ihre Korrektheit und Präzision überprüft sowie die Signatur gegebenenfalls korrigiert. Die Modellierung des Signaturfragments, dass die Überführung eines Programms in den verwundbaren Zustand erkennt, ist dabei besonders aufwendig. Abb. 1 verdeutlicht dieses Problem an einem Beispiel. Der Quelltext beschreibt einen verwundbaren Abschnitt des Gerätedateisystemmanagers *udev* (Version 124) von Linux, der die nicht-autorisierte Erlangung von Administrationsrechten ermöglicht. Die Ereignisse, die ein typischer Host-IDS-Sensor protokollieren kann (Betriebssystem- und Bibliotheksaufrufe), sind fett dargestellt. Bei Ausführung der dargestellten Programmfunktionen entsteht eine zeitlich geordnete Sequenz beobachtbarer Auditereignisse. Das IDS muss auf Basis dieser Ereignisse und mittels der Signatur (EDL-Notation [MSK05] in Abb. 1) erkennen, ob das Programm in einen verwundbaren Zustand überführt wird. Bei der manuellen Signaturmodellierung entstehen oftmals Signaturen, die unpräzise spezifiziert sind und deshalb auch Programmabläufe erkennen, die ein zum verwundbaren Programmabschnitt vergleichbares Verhalten aufweisen, jedoch keine Verwundbarkeit beinhalten. Diese Ungenauigkeiten führen zu Fehlalarmen, die durch eine Automatisierung des Signaturgenerierungsprozesses vermieden werden sollen.

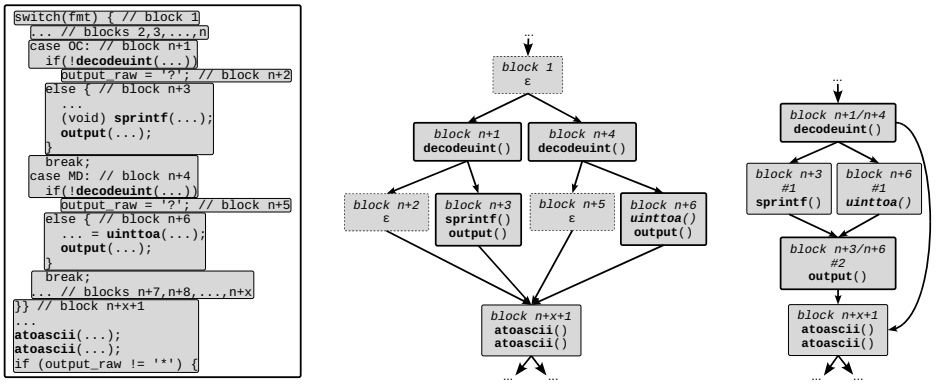
Der in diesem Beitrag vorgestellte automatisierte Signaturgenerierungsprozess lässt sich in die fünf Phasen Verwundbarkeitsanalyse, Programmmodellgenerierung, Programmmodellreduktion, Signaturgenerierung und Präzisionsabschätzung gliedern, die in den folgenden Unterabschnitten im Detail erläutert werden. Der Signaturmodellierer ist lediglich in der ersten und letzten Phase involviert.

2.1 Verwundbarkeitsanalyse

Ein erfolgreicher Angriff auf eine Verwundbarkeit erfordert in der Regel die Kompromittierung des Kontroll- oder Datenflusses der verwundbaren Anwendung mittels fehlerhafter Daten (bspw. durch Versenden von Nachrichten oder direkte Nutzereingaben). In einigen Fällen werden diese eingebrachten Daten auf ihre Integrität überprüft und die Verarbeitung im Fehlerfall abgebrochen. Infolgedessen verzweigen meist einige Kontrollflüsse in Programmabschnitte zur Fehlerbehandlung, in denen eine Ausnutzung der Verwundbarkeit nicht mehr möglich ist. Die verbleibenden Kontrollflüsse erreichen jedoch meist Programmstellen, an der die Verwundbarkeitsausnutzung unmittelbar bevorsteht oder bereits eingeleitet wird. Die entsprechenden Programmstellen werden im Folgenden als Punkte der *Verwundbarkeitsausnutzung* bezeichnet. Alle Programmkontrollflüsse, die durch einen Verwundbarkeitsausnutzungspunkt verlaufen, identifizieren gemeinsam den verwundbaren Programmabschnitt. Anhand dieser Kontrollflüsse und den daraus resultierenden Auditereignissen kann die Überführung eines Programmes in den verwundbaren Programmzustand durch ein IDS verfolgt werden. Für den weiteren Signaturgenerierungsprozess müssen demzufolge die Verwundbarkeitsausnutzungspunkte manuell festgelegt werden, was für den Signaturmodellierer nur mit geringem Aufwand verbunden ist, da sie in den veröffentlichten Sicherheitsmeldungen der Bug-Tracker oder CVE-Reports zumeist konkret benannt werden.

2.2 Programmmodellkonstruktion

Im ersten automatisierten Schritt des Signaturgenerierungsprozesses wird ein Modell des Programmverhaltens generiert. Als Ausgangsbasis dient der Programmquelltext oder der Binärcode der verwundbaren Anwendung. Die Konstruktion von Kontrollflussgraphen aus Binärcode wird bereits in [Fla04] erwähnt, kann aber bedingt durch Probleme bei der Auflösung von Funktionszeigern hier nicht verwendet werden. In Abb. 2a ist ein Beispiel für den Quelltext einer Programmfunktion dargestellt. Eine Funktion besteht aus einer Sequenz von Anweisungen, die den Kontrollfluss (bspw. Schleifen und Verzweigungen) sowie den Datenfluss (Variablenzuweisungen) eines Programms beeinflussen. Diese Anweisungssequenzen können in *Basisblöcke* mit jeweils einer einzigen Kontrollflussanweisung (typischerweise am Ende des Blocks) unterteilt werden. Nach der vollständigen Unterteilung einer Funktion in Basisblöcke (in Abb. 2a grau hinterlegt) wird aus diesen Kontrollflussanweisungen die Ausführungsreihenfolge der Basisblöcke hergeleitet. In ihrer Gesamtheit beschreiben die Vorgänger- und Nachfolgerbeziehungen zwischen den Basisblöcken einen lokalen *Kontrollflussgraph* für die untersuchte Funktion. Die Basisblöcke werden in den Knoten abgebildet und die Nachfolgerbeziehungen werden durch gerichtete Kanten dargestellt. Der Kontrollflussgraph für die Funktion der Abb. 2a ist in Abb. 2b dargestellt. Die generierten Kontrollussgraphen beschreiben zunächst nur das lokale Verhalten einzelner Programmfunktionen. Für eine Beschreibung des globalen Programmverhaltens müssen die in den lokalen Kontrollflussgraphen enthaltenen programminternen Funktionsaufrufe analysiert werden. Eine sofortige Integration der Kontrollfluss-



(a) Quellprogramm (b) Kontrollflussgraph (c) Reduktion

Abbildung 2: Generierung des Programmmodells

graphen aller aufgerufenen Funktionen erschöpft jedoch in vielen Fällen die vorhandenen Speicherressourcen. Demzufolge sind zunächst Reduktionsmaßnahmen notwendig.

2.3 Programmmodellreduktion

Die Reduktion des Programmmodells erfolgt in zwei Phasen. In der ersten Phase werden für typische IDS-Sensoren nicht sichtbare Kontrollflüsse entfernt und verzweigende Kontrollflüsse teilweise zusammengeführt. Die zweite Phase führt die Kontrollflussgraphen zur Beschreibung des lokalen Funktionsverhaltens zu einem globalen Kontrollflussgraph des Anwendungsverhaltens zusammen. Bei der Umsetzung der ersten Phase (*Reduktionsphase*) werden zunächst die für einen IDS-Sensor nicht sichtbaren Anweisungen aus den Basisblöcken der Kontrollflussgraphen entfernt. Falls ein Basisblock nach dieser Reduktion keine Anweisungen mehr enthält, wird er aus dem Kontrollflussgraph entfernt und die Vorgänger dieses Blockes werden mit dessen Nachfolgerblöcken verbunden. Anschließend wird der Kontrollflussgraph nach für IDS-Sensoren nicht zu unterscheidenden Verzweigungen durchsucht. Wenn zwei verzweigende Kontrollflüsse jeweils mit identischen Anweisungen beginnen, ist für ein IDS bei der Beobachtung der resultierenden Betriebssystem- oder Bibliotheksaufrufe zunächst nicht erkennbar, welcher Zweig ausgeführt wird. Kontrollflüsse dieser Art können reduziert werden, indem die identischen Anweisungssequenzen am Anfang der Zweige in einen gemeinsamen Basisblock zusammengefasst werden. In den Abbildungen 2b und 2c ist der beschriebene Reduktionsprozess an einem Beispiel dargestellt. Die Anweisungsblöcke der Abb. 2b enthalten ausschließlich von IDS-Sensoren protokollierbare Anweisungen. Basisblöcke, die keine protokollierbaren Anweisungen enthalten, sind mit gestrichelten Linien umrandet. Die beschriebenen Reduktionen überführen den Kontrollflussgraph der Abb. 2b in den in Abb. 2c dargestellten Kontrollflussgraph. Die Reduktion der Kontrollflussgraphen auf protokol-

lierbare Anweisungen führt dazu, dass in den nachfolgenden Generierungsschritten deutlich weniger Kontrollflüsse analysiert werden müssen.

In der zweiten Phase (*Konstruktionsphase*) werden die lokalen Kontrollflussgraphen der Programmfunktionen zu einem einheitlichen (flachen) Kontrollflussgraph vereint, der das globale Programmverhalten beschreibt. Hierfür müssen sämtliche Anweisungen, die programminterne Funktionen aufrufen, durch Kopien der Kontrollflussgraphen der aufgerufenen Funktionen ersetzt werden. Diese Integration (Abflachung der Aufrufhierarchie) wird wiederholt bis ein Kontrollflussgraph ohne programminterne Funktionsaufrufe entsteht. Um den Ressourcenbedarf (Arbeitsspeicher) der Konstruktionsphase zu minimieren, wird die Reduktionsphase für jeden in dieser Phase entstehenden Kontrollflussgraph ausgeführt. Bei der praktischen Realisierung der Kontrollflussgraphintegration muss zusätzlich beachtet werden, das die programminternen Funktionsaufrufe ebenfalls rekursiv erfolgen können. In einer prototypischen Implementierung der Konstruktionsphase wurden verschiedene Rekursionsprobleme behandelt, die hier jedoch aus Platzgründen nicht weiter erläutert werden. Zuletzt verbleibt ein einzelner Kontrollflussgraph, der das globale, von einem IDS-Sensor protokollierbare Programmverhalten beschreibt. Aus diesem Kontrollflussgraph wird im nächsten Schritt des Generierungsprozesses die Signatur abgeleitet.

2.4 Signaturgenerierung

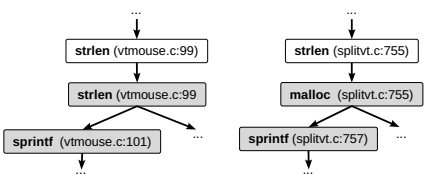
Die Signaturableitung verfolgt das Ziel, die durch die Verwundbarkeitsausnutzungspunkte verlaufenden Programmkontrollflüsse zu identifizieren, da diese einen potenziellen Angriff auf die Anwendung repräsentieren. Gleichzeitig muss sichergestellt werden, dass die generierte Signatur ausschließlich den verwundbaren Programmabschnitt erkennt und nicht die Ausführung anderer Programmabschnitte identifiziert. In Abb. 3 ist der Generierungsprozess für eine Signatur an einem Beispiel dargestellt. Der Quelltextauszug (3a) beschreibt eine Sicherheitlücke im Terminalprogramm *splitvt* [CVE10a], die eine Anhebung von Nutzerprivilegien ermöglicht. Im oberen Abschnitt der Abb. 3a ist der verwundbare Programmabschnitt dargestellt. Das letzte für ein typisches IDS wahrnehmbare Ereignis in diesem Abschnitt ist der Aufruf von *sprintf()*. Im unteren Abschnitt der Abb. 3a ist ein anderer, nicht-verwundbarer Programmabschnitt dargestellt, der ebenfalls einen Aufruf von *sprintf()* enthält. Die Kontrollflüsse, die zum ersten *sprintf()*-Aufruf führen, müssen eindeutig von den Kontrollflüssen, die zum zweiten *sprintf()*-Aufruf führen, unterschieden werden. Zu diesem Zweck werden zwei Mengen eingeführt. Die *Vulnerabilitätsmenge (*Vul*)* enthält alle zu den Verwundbarkeitsausnutzungspunkten führenden Kontrollflüsse. Die *Vergleichsmenge (*Vgl*)* enthält die Kontrollflüsse aus den nicht-verwundbaren Programmabschnitten, die bei einer Beobachtung durch einen IDS-Sensor von den verwundbaren Kontrollflüssen nicht zu unterscheiden sind. Beide Mengen werden mit den Basisblöcken der entsprechenden Kontrollflüsse initialisiert (Bsp. in Abb. 3c).

Der eigentliche Generierungsprozess läuft iterativ ab. Solange die Vergleichsmenge nicht leer ist, werden die Kontrollflüsse beider Mengen schrittweise verlängert, mit dem Ziel die Kontrollflusspfade der Vulnerabilitätsmenge von den Kontrollflusspfaden der Vergleichsmenge unterscheiden zu können. Dementsprechend werden die Elemente beider Mengen

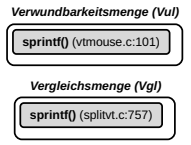
```
vtmouse.c:99-102, verwundbar:
if((strlen(...)+strlen(...)+1) > 512 )
  goto NoTitle;
sprintf(buffer, "xprop -id %s", title);
if((pipe=safe_popen(buffer, "r")) ) { ...

splitvt.c:755-757, nicht-verwundbar:
if((... malloc(strlen(...)+2)) == NULL)
  return(-1);
sprintf(temp, "-%s", args[0]);
```

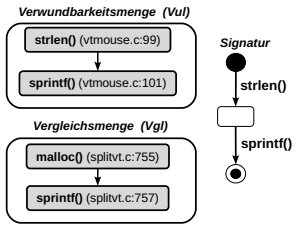
(a) Quellprogramm (Auszug)



(b) Kontrollflussgraphen (Auszug)



(c) Schritt 1 - Erzeugung der Startmengen



(d) Schritt 2 - Signaturgenerierung

Abbildung 3: Generierung der Signatur

als Untergraphen des globalen Kontrollflussgraph interpretiert. Da die Menge *Vul* mit den Verwundbarkeitsausnutzungspunkten initialisiert wurde, müssen die darin beschriebenen Kontrollflüsse entgegen der ursprünglichen Programmausführungsrichtung verlängert werden. Dementsprechend werden bei jeder Kontrollflussverlängerung für jeden Knoten der in diesen Mengen betrachteten Kontrollflussgraphausschnitte die zugehörigen Vorgängerknoten im Programmmodell ermittelt und in die Kontrollflussgraphausschnitte aufgenommen. Jeder Kontrollfluss wird somit um genau einen Basisblock erweitert. Die Abbildungen 3b und 3d zeigen die in der Vulnerabilitätsmenge und der Vergleichsmenge betrachteten Kontrollflussgraphaussschnitte (des globalen Kontrolflussgraphen) nach der ersten Verlängerung. Anschließend werden die Elemente beider Mengen paarweise miteinander verglichen. Falls zwei Kontrollflussgraphausschnitte mindestens einen identischen Kontrollfluss enthalten, werden sie als nicht unterscheidbar bewertet. Während des Vergleichsprozesses wird eine Statistik über identische und unterscheidbare Kontrollflusspfade geführt, die nach dem Abschluss des Generierungsprozesses für die Präzisionsabschätzung der Signatur genutzt wird. Aus der Vergleichsmenge werden alle Kontrollflussgraphausschnitte entfernt, die eindeutig von den Kontrollflussgraphausschnitten der Vulnerabilitätsmenge zu unterscheiden sind. Dieser Ablauf wird wiederholt, bis die Vergleichsmenge leer ist oder eine weitere Vergrößerung der Kontrollflussgraphausschnitte nicht mehr möglich ist. Im dargestellten Beispiel ist die Signaturgenerierung bereits nach der ersten Verlängerung abgeschlossen.

Im Anschluss an diesen Generierungsprozess wird die Vulnerabilitätsmenge in eine Signatur überführt, die typischerweise durch einen endlichen Zustandsautomaten beschrieben wird. Die Anweisungen in den Basisblöcken der Kontrollflussgraphausschnitte werden in Zustandsübergänge überführt und die Kanten der Kontrollflussgraphen werden in Zwischenzustände überführt. Anschließend wird ein Start- und Endzustand hinzugefügt. Die

jeweils erste Anweisung eines Kontrollflusses wird mit diesem Startzustand verbunden und die jeweils letzte Anweisung eines Kontrollflusses wird mit dem Endzustand verbunden. Im Beispiel der Abb. 3d ist eine Signatur abgebildet, die die Überführung von *splitvt* in den verwundbaren Programmzustand erkennen kann.

2.5 Präzisionsabschätzung

Nach dem Abschluss der Signaturgenerierung kann der Fall auftreten, dass der verwundbare Programmzustand nicht eindeutig zu identifizieren ist und die Vergleichsmenge nicht leer ist. Für diesen Fall wird eine Statistik geführt, die dem Signaturmodellierer die Abschätzung einer sinnvollen Kontrollflusspfadlänge und somit Signaturgröße ermöglicht. Die Präzisionsabschätzung basiert auf den Ergebnissen der Kontrollflusspfadvergleiche zwischen der Vulnerabilitätsmenge und der Vergleichsmenge. Diese Ergebnisse werden mittels der folgende Formel bewertet: $P = 1 - \#eq \,/\, (\#eq + \#diff)$.

In der dargestellten Formel entspricht *#eq* der Anzahl der in beiden Mengen identischen Kontrollflusspfade und *#diff* der Anzahl der zwischen beiden Mengen unterscheidbaren Kontrollflusspfade. Die Präzisionsabschätzung wird auf den Wertebereich [0,1] normalisiert. Bei einem Ergebnis von $P = 0$ ist die Signatur identisch zur Vergleichsmenge (geringste Präzision). Ein Ergebnis von $P = 1$ bedeutet, dass die Vulnerabilitätsmenge eindeutig von der Vergleichsmenge zu unterscheiden ist (höchste Präzision). In diesem Fall wird die resultierende Signatur keine Ungenauigkeiten bezüglich der Identifizierung des verwundbaren Programmzustandes aufweisen. Wenn die Analyse, bedingt durch die Programmstruktur, zu einem Ergebnis von $P < 1$ führt, kann der Signaturmodellierer einen Schwellwert d definieren, der eine maximal zulässige Abweichung festlegt.

Die Präzisionsbewertung beginnt nach der Erzeugung der beiden Mengen und wird nach jedem Signaturvergrößerungsschritt verfeinert, indem im Anschluss an den Mengenvergleich wieder die oben beschriebene Formel angewendet wird. Wenn die Präzisionsbewertung einen Wert von $P \geq 1 - d$ erreicht, wird die Vulnerabilitätsmenge gesichert. Diese Sicherung wird für die Generierung einer weniger präzisen Signatur genutzt, falls der Signaturgenerierungsprozess keine eindeutige Signatur generiert.

3 Anwendungsbeispiele

Um einen Eindruck über die Eignung des vorgestellten Verfahrens zu erhalten, wurde der gesamte Konstruktions- und Generierungsprozess in einem Werkzeug für C-Programme prototypisch umgesetzt und auf einem Testsystem (Quad-Core 2,83GHz) mehrfach exemplarisch evaluiert. Das implementierte Werkzeug automatisiert alle Schritte, von der Konstruktion des Programmmodells, über die Reduktion, bis hin zur Generierung der Signaturen und der Präzisionsabschätzung. Für die Evaluierung des Verfahrens wurden exemplarische Vertreter aus den Verwundbarkeitsberichten in [GLS10] ausgewählt. Dabei wurden unter anderem die folgenden zwei Beispiele evaluiert, die in diesen Berichten als kri-

tisch eingestuft wurden. Die aus der Evaluierung gewonnenen Ergebnisse sind vielversprechend.

3.1 Shadow – Anhebung von Nutzerprivilegien (CVE-2008-5394)

Im *login*-Programm (*shadow*-Version 4.0.18.1) von Linux, das für die Nutzerauthentifizierung und das Aufsetzen der Shell-Umgebung zuständig ist, existiert eine Verwundbarkeit, die das Überschreiben beliebiger Dateien erlaubt. Folgende Sicherheitswarnung wurde zum Zeitpunkt der Veröffentlichung der Verwundbarkeit herausgegeben: *„/bin/login in shadow 4.0.18.1 in Debian GNU/Linux, and probably other Linux distributions, allows local users in the utmp group to overwrite arbitrary files via a symlink attack on a temporary file referenced in a line (aka ut_line) field in a utmp entry.“* [CVE10c]

Bestimmung der Verwundbarkeitsausnutzung: Für die Ermittlung der Programmstellen, an denen eine Ausnutzung der Verwundbarkeit möglich ist, müssen lediglich die Hinweise in der Sicherheitswarnung ausgewertet werden. Die Sicherheitswarnung nennt konkret das verwundbare Programm (*/bin/login*), die Art des Angriffes (*symlink attack*) und ein Feld mit der Bezeichnung *ut_line*, das ein Bestandteil der utmp-Datenstruktur ist. Die Stelle der Verwundbarkeitsausnutzung wird nicht explizit erwähnt. Infolgedessen muss die Verwendung des *ut_line*-Feldes weiterverfolgt werden. Eine Durchsuchung der Programmquelltexte nach der Zeichenkette *ut_line* ermittelt in der Datei *login.c* den einzigen Verweis auf dieses Feld. Im *ut_line*-Feld der *utmp*-Datenstruktur wird der Pfad zur Gerätedatei des aktuell verwendeten Terminals hinterlegt. Die weitere Durchsuchung der Datei *login.c* nach der Verwendung dieses Feldes führt zu der Funktion *chown_tty()*, die die Zugriffsrechte des aktuell verwendeten Terminals ändert. Ein Angreifer kann über eine Manipulation dieses Feldes (und mit Hilfe einer symlink-Attacke) sicherstellen, dass die Zugriffsrechte einer beliebigen Datei auf die Zugriffsrechte der Gruppe *utmp* geändert werden. Der verwundbare Programmabschnitt endet mit dem Aufruf von *chown()* innerhalb der Funktion *chown_tty()*. Der Aufruf von *chown()* führt die zuletzt beschriebene Änderung der Zugriffsrechte aus und ist demzufolge mit der Ausnutzung der Verwundbarkeit gleichzusetzen.

Konstruktion und Reduktion des Programmmodells (Dauer: 1,8s): Das Programmmodell von *shadow* umfasst 305 Graphen mit insgesamt 548 Systemaufrufen und 3192 Bibliotheksaufrufen, die jeweils einzelne Programmfunktionen beschreiben. Durch die Integration und Reduktion dieser Funktionen wird der globale Kontrollflussgraph der *login*-Anwendung auf 111 Systemaufrufe und 539 Bibliotheksaufrufe reduziert.

Generierung des Signaturfragmentes und Präzisionsabschätzung: Obwohl eine einfache Zeichenkettensuche sechs Aufrufe von *chown()* innerhalb von *shadow* identifiziert, wird lediglich eine einelementige Signatur generiert: *chown()*. Die Code-Analyse identifiziert den *chown()*-Aufruf innerhalb der Datei *chowntty.c* als den einzigen Aufruf dieser Art in der *login*-Anwendung. Die anderen *chown()*-Aufrufe sind entweder ein Bestandteil anderer Hauptprogramme oder nicht erreichbar. Infolgedessen ermittelt der Prototyp bei der Präzisionsabschätzung einen Wert von $P = 1$, der gleichbedeutend mit einer eindeutigen Identifizierbarkeit des verwundbaren Programmabschnittes ist. Für die Erkennung des zu-

vor beschriebenen Angriffes ist die generierte Signatur jedoch nicht ausreichend, da der verwundbare Programmabschnitt ebenfalls bei legitimer Programmnutzung durchlaufen wird. Der Signaturmodellierer muss zusätzlich einen Vergleich des aktuell verwendeten Terminals mit dem im *ut_line*-Feld referenzierten Terminal ergänzen (bspw. über Analyse des *chown()*-Aufrufparameters).

3.2 ClamAV – DOS in rekursivem Aufrufpfad (CVE-2008-5314)

Die Antivirenlösung *ClamAV* enthält eine Verwundbarkeit, die in der folgenden Sicherheitswarnung beschrieben wurde: *„Stack consumption vulnerability in libclamav/special.c in ClamAV before 0.94.2 allows remote attackers to cause a denial of service (daemon crash) via a crafted JPEG file, related to the cli_check_jpeg_exploit, jpeg_check_photoshop, and jpeg_check_photoshop_8bim functions.“* [CVE10b]

Bestimmung der Verwundbarkeitsausnutzung: In der Meldung ist der vollständige rekursive Aufrufpfad aufgeführt, der zur Ausnutzung der Sicherheitslücke führt. Die Funktion *cli_check_jpeg_exploit* ruft zunächst die Funktion *cli_check_photoshop* auf. Im Anschluss erfolgt ein Aufruf der Funktion *cli_check_photoshop_8bim*, die wiederum die Funktion *cli_check_jpeg_exploit* aufruft (indirekte Rekursion). Der Zyklus beginnt demzufolge mit dem zuletzt genannten Aufruf innerhalb der Funktion *cli_check_photoshop_8bim* (Punkt der Verwundbarkeitsausnutzung). Unmittelbar vor diesem Aufruf erfolgt ein Betriebssystemaufruf von *lseek*, der von einem typischen IDS-Sensor protokolliert werden kann und infolgedessen als Ansatzpunkt für den Signaturgenerierungsprozess genutzt wird.

Konstruktion und Reduktion des Programmmodells (Dauer 42,3s): Das Programmmodell von *ClamAV* umfasst 931 Funktionen mit insgesamt 1335 Betriebssystemaufrufen und 4962 Bibliotheksaufrufen. Im Gegensatz zum ersten Beispiel hat der (reduzierte) globale Kontrollflussgraph des Virenscanners mit 3482 Betriebssystemaufrufen und 56448 Bibliotheksaufrufen einen gegenüber den summierten Einzelwerten größeren Umfang.

Generierung des Signaturfragmentes und Präzisionsabschätzung: Wegen der Programmstruktur von *ClamAV* sind die verwundbaren Programmabschnitte von den Vergleichsabschnitten kaum zu unterscheiden. Infolgedessen kann im *ClamAV*-Beispiel keine eindeutige Signatur generiert werden. Die Kontrollflusspfade werden im Signaturgenerierungsprozess bis zum Hauptprogramm verlängert, ohne ein unterscheidbares Ergebnis zu generieren. Der Signaturmodellierer muss in diesem Fall eine sinnvolle Signaturgröße wählen, die auf den Ergebnissen der Präzisionsabschätzung basiert. Die Tabelle 1 enthält einen Auszug aus den Ergebnissen dieser Präzisionsabschätzung. Im Vergleich zu der Anzahl der identischen Kontrollflüsse (*#eq*) wächst die Anzahl der eindeutig zu unterscheidenden Kontrollflüsse (*#cmp – #eq*) deutlich schneller. Für Kontrollflüsse mit zwei oder mehr Funktionsaufrufen konvergiert P bereits gegen die obere Grenze ($P = 1$). Wenn der Signaturmodellierer einen Schwellwert $d = 0.03$ für die Abweichung von der oberen Grenze festlegt, wird der in Abb. 4 dargestellte Untergraph generiert. Bei der praktischen Anwendung der aus diesem Graph generierten Signatur können Fehlalarme nicht gänzlich ausgeschlossen werden. Vom Signaturmodellierer sind keine weiteren Änderungen an der

Signatur notwendig, da die Identifizierung des rekursiven Aufrufpfades bereits den Angriff selbst identifiziert. Die Präzisionsabschätzung deutet an, dass die Fehlalarmrate der generierten Signatur als eher gering einzuschätzen ist.

Pfadlänge	1	2	3	4	5	6
Pfadvergleiche (*#cmp*)	1046	13541	151340	368896	1148018	2764263
Übereinstimmung (*#eq*)	1046	1222	13565	10881	12234	16946
Präzision (P)	0.000	0.901	0.910	0.971	0.989	0.994

Tabelle 1: Präzsionsabschätzung

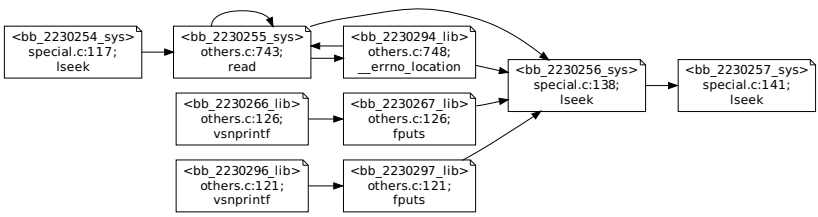


Abbildung 4: Kontrollflussgraphausschnitt der Pfadlänge 4

4 Zusammenfassung und Ausblick

Der in diesem Beitrag vorgestellte Signaturableitungsprozess generiert aus dem Quelltext verwundbarer Anwendungen automatisiert Signaturen für Host-basierte IDS. Diese Signaturen können die Überführung einer Anwendung in einen verwundbaren Programmzustand erkennen. Zu Beginn des Generierungsprozess werden lediglich wenige Informationen über verwundbare Programmstellen benötigt, die in fast allen Verwundbarkeitsberichten und Bug-Trackern verfügbar sind. Eine Evaluierung des Prozesses mittels realer Verwundbarkeiten hat zusätzlich gezeigt, dass die Präzisionsabschätzung eine Festlegung von sinnvollen Signaturgrößen ermöglicht, wenn Fehlalarme nicht gänzlich vermieden werden können. Die in diesem Beitrag exemplarisch evaluierten Verwundbarkeiten deuten desweiteren an, dass typischerweise kleine Signaturgrößen zu erwarten sind.

Eine Generierung von Signaturen, für die Fehlalarme völlig ausgeschlossen werden können, erfordert weiterführende Untersuchungen. Offen ist zum Beispiel, ob eine zusätzliche Datenflussanalyse im Programmquelltext die Genauigkeit der generierten Signaturen verbessert. In einem nächsten Schritt könnte ein IDS implementiert werden, das auf den Prinzipien schneller Sandbox-Verfahren wie z.B. *sydbox* [syd10] basiert. In Sandbox-basierten IDS können Angriffe unterbrochen werden, indem die Ausnutzung der Verwundbarkeit verhindert wird.

Literatur

[BNS+06] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *IEEE Symposium on Security and Privacy*, pages 2–16, CA, USA, 2006. IEEE Computer Society.

[CCZ+07] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the 21st ACM Symposium on Operating systems principles*, pages 117–130, USA, 2007. ACM.

[CPWL07] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael E. Locasto. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *IEEE Symposium on Security and Privacy*, pages 252–266, Berkeley, CA, 2007. IEEE Computer Society.

[CVE10a] Common Vulnerabilities and Exposures – CVE-2008-0162. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-0162`, January 2010.

[CVE10b] Common Vulnerabilities and Exposures – CVE-2008-5314. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-5314`, January 2010.

[CVE10c] Common Vulnerabilities and Exposures – CVE-2008-5394. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-5394`, January 2010.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996. IEEE Computer Society.

[Fla04] Halvar Flake. Structural Comparison of Executable Objects. In Ulrich Flegel and Michael Meier, editors, *DIMVA*, volume 46 of *LNI*, pages 161–173. GI, 2004.

[GJM04] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, USA, 2004. The Internet Society.

[GLS10] Gentoo Linux Security Advisories. `http://www.gentoo.org/security/en/glsa/index.xml`, January 2010.

[MSK05] Michael Meier, Sebastian Schmerl, and Hartmut König. Improving the Efficiency of Misuse Detection. In *DIMVA 2005*, volume LNCS 3548, pages 188–205, Vienna, Austria, July 2005. Springer.

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, CA, USA, 2005. IEEE Computer Society.

[SBD01] R. Sekar, M. Bendre, and D. Dhurjati. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, CA, USA, 2001. IEEE Computer Society.

[syd10] sydbox. `http://projects.0x90.dk/wiki/sydbox`, February 2010.

[WD01] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, pages 156–168, Oakland, CA, USA, 2001. IEEE Computer Society.

[WFGPM09] T. Werner, C. Fuchs, E. Gerhards-Padilla, and P. Martini. Nebula - Generating Syntactical Network Intrusion Signatures. In *Proc. of the 4th International Conference on Malicious and Unwanted Software*, Montreal, Canada, October 13-14 2009.