Ein Überblick über statisches Analyse von User/Kernel Pointer Bugs mittels Typinferenz

Kristof Klee <n00bian@gmail.com>

19.12.2007

Zusammenfassung

Diese Arbeit bietet eine Zusammenfassung, der Arbeit von Rob Johnson und David Wagner [JW04], wie User/Kernel Pointer Bugs mittels Typinferenz, aufbauend auf dem Tool CQUAL, statisch ermittelt werden können.

Hierfür wurde CQUAL um einige Funktionalitäten erweitert, die erlauben mit weniger Annotationen die Analyse durchzuführen. Wichtige neue Funktionalitäten sind Kontextsensitivität, der bessere Umgang mit Strukturen und die bessere Analyse von Umwandlungen zwischen Zeiger und Integer. Als Test wurde diese Methode auf den Linux Kernel angewandt und so mehrere User/Kernel Pointer Fehler entdeckt und behoben.

Inhaltsverzeichnis

1 Einführung

Fast alle Programme müssen mit Daten umgehen, die von Benutzern beeinflusst werden könnten und somit potenziell unsicher sind. Eine Tatsache, die bereits zu Methoden geführt hat, diese unsicheren Eingaben und deren Verwendung mittels Tainting zu verfolgen [STFW01].

Besonders gefährlich ist der Umgang mit Zeigern aus dem User-Space im Betriebssystem Kernel, da hier eine falsche Verwendung dieser potenziell gefährlichen Zeiger das gesamte Betriebssystem abstürzen lassen könnte oder aber einem Angreifer Vollzugriff erteilen könnte.

Zwar bietet zum Beispiel der Linux Kernel spezielle Funktionen um den Umgang mit User Pointern zu erleichtern und sicher zu machen, der schiere Umfang der verschiedenen Treiber Module und Kontributoren führt jedoch dazu, dass User/Kernel Pointer Exploits immer wieder möglich sind. Generell ist der Umgang mit User Pointern in jedem Kernel eine potenzielle Angriffsfläche.

Ein Ansatz ist es, den Datenfluss mittels zusätzlicher Typannotationen und Typinferenz durch statische Analyse nachzuvollziehen . Diese Methode wurde für den Spezialfall der User/Kernel Pointer Bugs erweitert und verfeinert um mit weniger Annotationen weniger False Positives zu erzielen.

Hierfür wurde CQUAL um Kontextsensitivität, verbessertem Umgang mit C-Strukturen sowie genauere Analyse von Typumwandlungen zwischen Zeiger und Integer erweitert, die das Auftretten von False Positives um den Faktor 20 verringerten.

Der Ansatz mittels CQUAL ist somit anders als der MECA [YKXE03], dass versucht keine False Positives zu liefern, dafür aber einige False Negatives in kauf nimmt. CQUAL liefert keine False Negatives.

2 User/Kernel Pointer Bugs

Bei User/Kernel Pointer Bugs handelt es sich um eine Spezialform des normalen Problems, mit dem Umgang von Daten, die von Benutzern beeinflusst werden könnten.

Die Besonderheit hierbei ist, dass ein falscher Umgang mit Zeigern, die an den Kernel von Programmen übergeben werden, das gesamte System abstürzen lassen, dem Programm besondere Rechte geben oder aber spezielle Daten zugänglich machen könnten, die dem Benutzer ansonsten verwährt blieben.

Diese Zeiger müssen vom Angreifer natürlich besonders präpariert werden.

2.1 der Linux Kernel als Beispiel

Als Beispiel wird der Linux Kernel genannt, der in seiner 32-Bit Version den Kernel Speicher in das oberste Gigabyte virtuellen Speichers des Programms mapped.

Somit muss bei einem Systemaufruf nur der Lese/Schreib-Schutz über diesen Speicherbereich aufgehoben und beim Zurückkehren in das Programm wieder hergestellt werden.

Bieten spezielle Kernelfunktionen nun die Möglichkeit Zeiger als Parameter zu übergeben um Daten von diesem Speicherbereich zu lesen oder hineinzuschreiben, ist es einem Angreifer möglich mittels ungültiger Adressen den Kernel abstürzen zu lassen.

Weiters könnte mittels Übergabe von Adressen, die im Kernel Speicherbereich liegen:

- Code im Kernel durch eigenen ersetzt werden um so zum Beispiel Sicherheitsabfragen zu umgehen
- Benutzerrechte beeinflusst werden um Administratorrechte zu erhalten
- oder geheime Daten wie zum Beispiel Schüssel oder Passwörter ihren Weg in den Speicherbereich des Programms finden.

3 CQUAL

CQUAL wurde entwickelt, um C Programme mittels zusätzlicher Type Qualifier auf spezielle Typfehler zu überprüfen (siehe [STFW01]). Hierfür wird der Datenfluss von qualifizierten Typen mittels Typinferenz durch das Programm verfolgt und gegebenenfalls ein Typfehler geworfen.

Die Analyse ist statisch und kann während der Entwicklung im Zuge der Übersetzung stattfinden. Benötigt wird der Prekompilierte Quellcode, Definitionsdateien für die verwendeten Type Qualifier und zusätzliche Headerdateien in denen Schlüsselfunktionen annotiert sind um die Typinferenz nutzen zu können.

Die Vorteile dieses Ansatzes sind:

- wenige Annotationen sind nötig um den gesamten Quellcode abzudecken
- Typfehler können während der Entwicklung beim Kompilieren entdeckt werden
- Typinferenz mittels Type Qualifier ist stichhaltig, dh. wenn keine Fehler gefunden werden existieren auch keine

4 Type Qualifiers

Zum Entdecken von User/Kernel Pointer Bugs wurden die Qualifier *user* und *kernel* definiert, wobei *kernel* ein Subtyp von *user* ist ($kernel < user$). Dies bewirkt, dass Funktionen, die einen User Pointer erwarten auch einen Kernel Pointer akzeptieren können, ohne das CQUAL einen Typfehler meldet.

Da jeder Teil eines Typs annotiert werden kann ist es sehr wohl möglich Kernel Pointer auf User Daten zu haben als auch umgekehrt. Auch Schachtelungen sind möglich, so dass ein Kernel Pointer auf User Pointer ermöglicht wird.

5 Stichhaltigkeit

Damit die Analyse durch Typinferenz mittels Type Qualifier als stichhaltig betrachtet werden kann wurden einige Annahmen getroffen. Es wird davon ausgegangen:

- dass das Programm keine Buffer Overflows enthält

- dass Unions richtig benutzt werden
- dass das Programm als Ganzes übersetzt wird und nicht nur einzelne Module
- das Programm keinen Inline Assembler Code enthält

Die meisten dieser Bedingungen können mittels anderer Analysetools gewährleistet werden.

6 CQUAL Erweiterungen

Für die gegebene Problemstellung wurde CQUAL um einige Funktionalitäten erweitert, um sowohl die Benutzung zu erleichtern als auch das Auftreten von False Positives zu vermindern.

6.1 Kontextsensitivität

CQUALs Constraintsystem hat Probleme falls eine Funktion mehrmals mit verschieden qualifizierten Typen aufgerufen wird, da es von Haus aus keine Polymorphie unterstützt und somit vermehrt False Positives meldet.

Dieses Problem versuchte man schon auf andere Weise mittels polymorpher Annotationen zu beheben (siehe [STFW01]), was jedoch zusätzliche Annotationen und Syntax für Polymorphie erfordert.

Durch das Hinzufügen von Konstextsensitivität wird gewährleistet, dass nur jene Pfade im Constraintgraph verfolgt werden, die auch wirklich mit dem aktuellen Aufruf einhergehen. Dazu wurden die Klammerungen im Code durchnummeriert und die Kanten des Graphs jeweils mit den dazugehörigen Nummern benannt und gekennzeichnet, ob die Klammerung geöffnet oder geschlossen wurde.

Dadurch wurde diese Art der False Positives komplett eliminiert und zeigte bei Verwendung in anderen Anwendungen von CQUAL, zur Auffindung von Format String Fehlern, eine Reduktion von bis zu 90% der Falschmeldungen.

6.2 Umgang mit Strukturen

Bislang behandelte CQUAL alle Instanzen einer Struktur gleich. Dies führte dazu, das beim Verwenden verschiedener Instanzen der selben Struktur jeweils mit unterschiedlichen qualifizierten Typen, zwangsläufig ein Typfehler auftrat.

Um diese False Positives zu eliminieren wurde CQUAL so erweitert, dass es bei jeder Verwendung eins Strukturfeldes dieses individuell qualifiziert. Um den Speicherbedarf linear zu halten wird dies nur bei der wirklichen Verwendung eines Strukturfeldes gemacht und nicht automatisch für alle Feler jeder Instanz.

Außerdem werden bei Zuweisung von einer Strukturinstanz auf die andere alle Qualifier übernommen.

Diese Änderung betrifft auch Unions, da diese von CQUAL wie Strukturen behandelt werden.

Um den Spezialfall abzudecken, wenn Userdaten über eine Struktur kopiert werden und ein Feld dieser Struktur dann zum dereferenzieren benutzt wird, was potenziell zu einem User/Kernel Pointer Bug führt, da nun auch die Felder der Struktur unter der Kontrolle des Benutzers sind, wurde das Type Qualifier Regelsystem erweitert. Es kann nun auch die Beziehung zwischen dem Qualifier einer Struktur und deren Felder nachbilden und so wohlgeformte Constraints gewährleisten.

6.3 Analyse von Casts zwischen Pointer und Integer

Beim Casten zum Beispiel von einem Zeiger auf Zeiger zu einer Integer setzte CQUAL vorraus, dass alle Qualifier gleich sind. Das bedeutet sowohl der Type Qualifier des Zeigers als auch der Typ Qualifier des Zeigers auf Zeiger als auch der der Integer mussten gleich sein. Dies führte zu vielen False Positives und verletzte auch die Stichhaltigkeit der Analyse.

Dies wurde dadurch behoben, dass alle int in einem solchen Fall als typlose Zeiger behandelt werden. Es werden zwar die Type Qualifier tieferer Ebenen noch immer gleichgesetzt, die erste Ebene jedoch produziert einen richtigen Constraint, wodurch die Stichhaltigkeit garantiert und das Auftreten von False Positives vermindert wird.

7 Test am Linux Kernel

Um sowohl die Performance als auch den Nutzen in wirklichen Entwicklungszyklen zu testen wurde die Methode am Linux Kernel der Version 2.4.20 als auch 2.4.23 getestet.

Es wurden sowohl einzelne Module als auch der gesamte Kernel auf einmal analysiert.

Einzelne Module widersprechen zwar den Annahmen die getroffen wurden um Stichhaltigkeit zu garantieren, spiegeln aber wieder, wie Programmierer dieses System in Wirklichkeit während der Entwicklung benutzen würden. Hierfür wurde die Subtypbeziehung zwischen kernel und user aufgehoben um inkonsistente Zeigerverwendung aufzuzeigen, die Fehler produzieren könnte aber ohne Kontext des gesamten Quellcodes von CQUAL nicht entdeckt werden würde.

7.1 Ergebnisse

Insgesamt wurden in Kernel Version 2.4.20 7 Fehler gefunden, wobei CQUAL 275 Typfehler meldete. In Kernel Version 2.4.23 wurden 264 Typfehler geworfen, wobei 6 davon tatsächlichen Fehlern entsprachen.

Bei Analyse des gesamten Quellcodes auf einmal benötigte CQUAL 10GB Hauptspeicher und 90 Minuten auf einem Itanium Prozessor mit 800 MHz.

Ein Grossteil der False Positives war in den Treibern und nicht im Kern des Kernels zu finden.

8 Weiterführend

CQUAL wurde bereits benutzt:

- um Format String Fehler zu entdecken [STFW01]
- um Autorisierungspunkte im Linux Kernel zu verifizieren [ZEJ02]
- um private Daten aus Crash-Reporten zu entfernen [BHS03]
- um korrekte Benutzung von garbage collected „__init" Daten im Linux Kernel zu überprüfen [FJKA06]
- zum Auffinden von Y2K-Fehlern [EFA]

Alle vorgenommenen Verbesserungen wirken sich somit auch auf diese Applikationen aus, insofern sie die modifizierte CQUAL Version benutzen.

MECA [YKXE03], ein weiteres Tool zum finden von User/Kernel Pointer Bugs, versucht so wenig False Positives wie möglich zu generieren und liefert dafür aber False Negatives.

Kontrollfluss basierten Analysen wie zum Beispiel MOPS [CW02] oder SLAM [BR02] können CQUAL gut ergänzen.

9 Conclusio

Die vorgestellte Analyse liefert eine zuverlässige Methode User/Kernel Pointer Bugs zu entdecken und verbesserte CQUAL um Funktionalitäten die auch für andere Typ Qualifier und Data Flow basierte Analysen verwendbar sind.

Die Analyse größerer Systeme ist mit akzeptablem Aufwand möglich.

Um die Benutzbarkeit zu erhöhen müssen False Positives weiter verringert werden. Dazu muss in diesem Fall vor allem der Umgang mit Unions und Strukturen mittels Flow-sensitivität verfeinert werden.

Die Stichhaltigkeit der Analyse ist wichtig um den Ergebnissen vertrauen zu können. Im Vergleich zu MECA [YKXE03], dass bei späteren Linux Kernel Versionen Fehler nicht fand, die mittels CQUAL schon in früheren gefunden wurden, liefert dieser Ansatz vertrauenswürdige Ergenisse.

Schliesslich zeigt sich, dass mittels Typannotationen und statischer Analyse ein breites Spektrum an Sicherheitsproblemen schon beim Kompilieren entdeckt werden kann. Beispiele Hierfür liefern sowohl [JW04] als auch [STFW01].

Literatur

[BHS03] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: a system for generating secure crash information. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming langua-*

ges, pages 1–3, New York, NY, USA, 2002. ACM.

[CW02] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM.

[EFA] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. Carillon — a system to find Y2K problems in C programs.

[FJKA06] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[YKXE03] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334, New York, NY, USA, 2003. ACM.

[ZEJ02] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.