TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

185.307

Seminar aus Programmiersprachen

Dynamic Tainting mit Dytan

Johann Grabner

e0226102@student.tuwien.ac.at

6. Dezember 2007

Kurzfassung

Diese Arbeit fasst die wichtigsten Aspekte des Artikels [1] Dytan: A Generic Dynamic Taint Analysis Framework“ von James Clause, Wanchun Li und Alessandro Orso zusammen. Zu den Unterschieden anderer Arbeiten in diesem Themenbereich wird Stellung genommen.

Beim Dynamic Tainting werden Eingabedaten bestehender Programme zur Laufzeit mit einer oder mehreren Markierungen attributiert. Diese Kennzeichnungen werden an Daten weitergegeben, deren Inhalt durch die Eingabedaten mitbestimmt wird. Leitet sich das Verhalten einer Programmfunktion aus markierten Daten ab, können zusätzliche Aktionen gesetzt werden.

Ein Anwendungsgebiet indem Dynamic Tainting bereits oftmals eingesetzt wurde, ist die Erhöhung der Sicherheit von Programmen zur Laufzeit, indem beispielsweise sicherheitskritische Funktionen mit als unsicher markierten Parametern nicht ausgeführt werden.

Ein großer Teil der Anwendungen von Dynamic Tainting wurde jedoch für einen speziellen Problembereich konzipiert. Der behandelte Artikel stellt ein generelles Framework für Dynamic Tainting vor, um der Limitation auf einen bestimmten Anwendungsbereich zu entgehen.

Inhaltsverzeichnis

1 Einleitung..2
2 Verwandte und ähnliche Arbeiten...2
  2.1 Generelle Ansätze...3
  2.2 Erkennen und Abwenden von Angriffen...3
  2.3 Durchsetzung von Regeln für den Informationsfluss................................3

2.4 Software-Testen..3
3 Generelles Framework..4
3.1 Taint-Sources..4
3.2 Taint-Propagation..4
3.2.1 Identifizierung abgeleiteter Daten..5
3.2.2 Mapping-Funktionen..5
3.3 Taint-Sinks..5
4 Implementierung von Dytan..6
5 Evaluation von Dytan..6
6 Zusammenfassung..6
7 Referenzen..7

1 Einleitung

Der behandelte Artikel [1] stellt ein Framework für Dynamic Tainting vor. Beim Design des Frameworks wurde besondere Rücksicht drauf genommen, dass es allgemein einsetzbar bleibt, um die Einsatzmöglichkeiten nicht auf nur einen Problembereich wie zum Beispiel die Sicherheit von Applikationen zu beschränken. Folgende Punkte werden von den Autoren als Vorteile gegenüber bereits existierenden Ansätzen beschrieben:

- **Anpassbarkeit**
  Es lässt sich einfach spezifizieren, welche Daten beim Tainting berücksichtigt werden und wie diese markiert werden sollen. Daten können gleichzeitig mit mehreren unterschiedlichen Markierungen versehen werden. Die Art der Verbreitung von Taint-Markierungen und die Prüfung ist ebenfalls leicht anzugeben.
- **Tainting auf Basis von Daten- und Kontrollfluss**
  Bei der Taint-Analyse wird wahlweise auch Rücksicht auf den Kontrollfluss genommen. Daten, die in Bedingungen abgefragt werden, können andere Daten indirekt beeinflussen. Je nachdem welcher Pfad im Programm eingeschlagen wird, können unterschiedliche Zuweisungen bzw. Berechnungen stattfinden.
- **Analyse ausführbarer Programme**
  Das Framework operiert direkt auf ausführbaren binären Dateien und benötigt somit keinen Zugriff auf den Quelltext der Anwendungen. Das Framework stellt keine weiteren Anforderungen an seine Einsatzumgebung und ist somit leicht einzusetzen.

Im nächsten Abschnitt wird der behandelte Artikel mit anderen Arbeiten in diesem Themenbereich verglichen. Der dritte Abschnitt beschäftigt sich mit der allgemeinen konzeptionellen Beschreibung des entworfenen Frameworks. Dessen Implementierung „Dytan" wird im darauf folgenden fünften Abschnitt behandelt. Die Evaluation der vorgestellten Implementierung ist letztendlich Thema des sechsten Abschnitts.

2 Verwandte und ähnliche Arbeiten

Dieser Abschnitt führt einige andere Arbeiten zum Themenbereich „Dynamic Tainting" an und geht gegebenenfalls kurz auf etwaige Unterschiede zu [1] ein.

Gleichzeitig fungiert er als Überblick über mögliche Einsatzszenarien von Dynamic Tainting. Die Anwendungsbereiche reichen inzwischen längst über den Bereich der Applikationssicherheit, indem sich Dynamic Tainting bereits bewährt hat, hinaus.

2.1 Generelle Ansätze

In [2] wird ein Ansatz vorgestellt, indem C-Programme über die Verwendung eines eigens dafür angefertigten Compilers um Dynamic Tainting erweitert werden können. Hierzu muss jedoch der Quelltext neu übersetzt werden. Dytan ist hingegen direkt auf ausführbaren Dateien einsetzbar. Des Weiteren geben die Autoren von [1] an, dass es an der Unterstützung für Kontrollfluss basiertes Tainting fehlt.

Eine ähnliche Herangehensweise wird in [3] vorgestellt. Dort wird der Quelltext von C-Programmen in um Tainting erweiterten Quelltext transformiert. Diese Lösung ermöglicht, verglichen mit Dytan, keine multiplen Taint-Markierungen für Daten.

2.2 Erkennen und Abwenden von Angriffen

Dynamic Tainting wurde in vielen Arbeiten verwendet um Angriffe auf ein Software-System aufzuspüren und abzuwenden. Zu den häufig untersuchten Angriffsarten zählen Overwrite-Attacks (Überschreiben von Programmdaten) und Command Injections (Einschleusen von Kommandos).

TaintCheck, ein in [4] vorgestelltes Werkzeug, ist ähnlich zu Dytan, weil es ebenfalls auf ausführbaren Programmen operiert. Das Anwendungsgebiet von TaintCheck ist das Erkennen von Overwrite-Attacks, während Dytan hingegen vielseitiger einsetzbar ist.

Das Thema von [5] ist die Erkennung unsicherer Benutzereingaben in Java, die ein Sicherheitsproblem in der weiteren Ausführung des Programms darstellen können. Java-Klassen werden während der Laufzeit mit Tainting-Funktionalität instrumentiert. In [6] wird der Standard-PHP-Interpreter durch einen mit Tainting erweiterter ersetzt. Die Implementierungen aus [5] und [6] sind auf jeweils eine Programmiersprache limitiert.

2.3 Durchsetzung von Regeln für den Informationsfluss

Ein weiteres Einsatzszenario für Dynamic Tainting ist die Regelung des Informationsflusses. Greifen Systemteile auf Daten zu, ohne die entsprechende Berechtigung inne zu haben, kann dies durch an die Daten vergebene Taint-Markierungen erkannt und verhindert werden.

Als Beispiel für die Regelung des Informationsflusses kann man RIFLE [7] anführen. RIFLE transformiert binäre Programme in andere, die auf einer Prozessorarchitektur mit Unterstützung für sicheren Informationsfluss ausführbar sind.

2.4 Software-Testen

Auch im Software-Testen kann Dynamic Tainting von Nutzen sein. Durch den

Einsatz von Tainting kann beispielsweise ein Graph aufgebaut werden, der die Abhängigkeiten von Funktionsparametern und Bedingungen in Funktionen repräsentiert. Aus diesen Informationen lassen sich auf einfachem Wege Eingabedaten ableiten, um zum Beispiel beim Testen eine geforderte Anweisungsüberdeckung zu erreichen. Im COMET-System [8] werden solche Methoden für C-Programme angewandt.

3 Generelles Framework

Zur Durchführung einer Taint-Analyse müssen Sources, Propagation und Sinks spezifiziert werden. Dieser Abschnitt gibt einen Überblick darüber, welche Möglichkeiten das Framework hierzu bereit stellt.

3.1 Taint-Sources

Taint-Sources sind Progammdaten, die initial mit Taint-Markierungen attributiert werden sollen. Es folgt eine Aufzählung der Daten die sich im Framework als Taint-Sources spezifizieren lassen.

- **Variablen und Speicheradressen**
  Variablen werden über Namen und Gültigkeitsbereich als Taint-Sources angegeben. Bei Speicherbereichen erfolgt die Angabe über einen Offset relativ vom Programmanfang.
- **Returnwerte von Funktionen**
  Durch das Anführen des Funktionsnamen werden retournierte Daten mit Taint-Markierungen versehen.
- **Daten aus Streams**
  Hier gibt es zwei Möglichkeiten der Spezifikation. Zum Ersten können drei verschiedene Typen von Streams (Netzwerk, Dateisystem und Keyboard) als Sources deklariert werden. Die zweite Variante ist die Angabe eines speziellen Streams aus dem Netzwerk oder Dateisystem. Dies erfolgt über einen eindeutigen Bezeichner, der IP-Adresse oder dem absoluten Dateipfad.

Zusätzlich zur Deklaration der Taint-Sources muss definiert werden, mit welchen Taint-Markierungen aus einer Quelle stammende Daten versehen werden sollen. Das Framework unterstützt unterschiedliche und mehrere Markierungen pro Datenobjekt.

Zuletzt ist anzumerken, dass falls gewünscht unterschiedliche Markierungen für aus einem Stream gelesene Daten verteilt werden können. Die vergebenen Markierungen hängen dann von der Menge der bereits gelesenen Daten ab.

3.2 Taint-Propagation

Wie die Propagierung von Taint-Markierungen an abgeleitete Daten zu erfolgen hat, muss definiert werden. Hierzu muss ...

- eine Methode zur Erkennung abgeleiteter Daten gewählt
- eine sogenannte Mapping-Funktion definiert

werden. Beides wird in diesem Abschnitt erklärt.

3.2.1 Identifizierung abgeleiteter Daten

Das Framework stellt dem Benutzer folgende zwei Methoden zur Erkennung abgeleiteter Daten zur Verfügung:

- **Datenfluss basiert**
  Zur Ausbreitung von Taint-Informationen wird nur der Datenfluss herangezogen. Variablen gelten als abgeleitet, wenn deren Wert aus den Werten anderer Variablen berechnet wird.
- **Daten- und Kontrollfluss basiert**
  Zusätzlich zum Datenfluss wird der Kontrollfluss beim Tainting berücksichtigt. Variablen, die zum Auswerten von Bedingungen verwendet werden, geben ihre Taint-Information an eventuell in den jeweiligen Zweigen modifizierte Variablen weiter.

3.2.2 Mapping-Funktionen

Eine Mapping-Funktion wird ausgeführt, wenn eine Anweisung ein Ergebnis liefert, das auf mit Taint-Markierungen versehenen Daten basiert. Die Taint-Informationen aller beteiligten Daten werden als Parameter an die Mapping-Funktion übergeben. Der Funktionswert ergibt die Taint-Information des Anweisungsergebnisses.

Standardmäßig werden vom Framework die Mengen der Taint-Markierungen unterschiedlicher Daten vereinigt, sodass das Ergebnis alle Markierungen der Daten, von denen es abhängt, enthält. Der Benutzer kann jedoch auch eigene Mapping-Funktionen definieren.

3.3 Taint-Sinks

Unter einem Taint-Sink wird im Kontext des Frameworks eine Stelle im Programmcode verstanden, an welcher der Benutzer Taint-Prüfungen durchführen will. Jeder Sink hat eine eindeutige Nummer und jedem Sink sind eine oder mehrere benutzerdefinierte Überprüfungsfunktionen zugeordnet. Die Spezifikation eines Sinks ist durch die Angabe einer der beiden folgenden Alternativen vollständig:

- **Speicherort von Daten und Stelle im Code**
  Sobald das Programm die Stelle im Code erreicht, wird die Überprüfungsfunktion mit den zum Speicherort assoziierten Taint-Markierungen aufgerufen. Zum Angeben eines Speicherorts kann der Name der Variable inklusive Gültigkeitsbereich, ein Funktionsname zusammen mit dem Parameterindex, ein Funktionsname (für Returnwert) oder eine Adresse verwendet werden. Stellen im Code können Anfang oder Ende einer genannten Funktion oder eine Adresse im Programm sein.
- **Maschinenbefehltyp**
  Vor der Ausführung des angeführten Maschinenbefehls im Programm wird die Überprüfungsfunktion aufgerufen. Diese Funktion enthält als Parameter die Taint-Informationen der vom Maschinenbefehl gelesenen Daten.

4 Implementierung von Dytan

Dieser Abschnitt gibt einen Überblick über die Implementierung. Dytan im Einsatz ist auf Abbildung 1 ersichtlich.

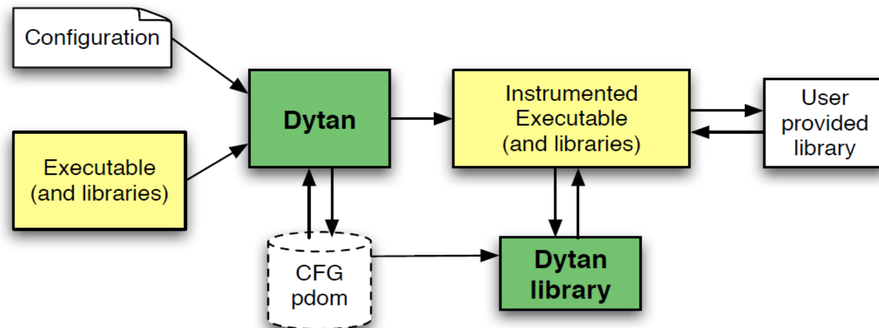

Abbildung 1: Dytan im Einsatz, Quelle [1]

Dytan arbeitet auf x86-Binaries. Die Konfiguration von Dytan erfolgt über eine XML-Datei. In dieser Konfigurationsdatei werden Taint Sources und Sinks zusammen mit der Propagation spezifiziert. Das ausführbare Programm wird samt seiner Bibliotheken während der Laufzeit um Dynamic Tainting erweitert. Das instrumentierte Programm benötigt während der Laufzeit Zugriff auf die Dytan-Bibliothek, welche die Propagierung von Taint-Informationen übernimmt, und auf die vom Benutzer bereitgestellte Bibliothek. Die Bibliothek des Benutzers enthält Funktionen zur Überprüfung von Taint-Markierungen und gegebenenfalls eigene Mapping-Funktionen. Falls die Daten- und Kontrollfluss basierte Tainting-Methode gewählt wurde, müssen zusätzliche Flussgraphen berechnet werden.

5 Evaluation von Dytan

Bei der von den Autoren von [1] durchgeführten Evaluation von Dytan stellte sich heraus, dass sich der Aufwand der Implementierung von bereits bestehenden Ansätzen zur Vermeidung von Overwrite-Attacks und SQL-Injections in Grenzen hielt.

Des Weiteren wurde gezeigt, dass signifikante Unterschiede in der Menge der gesammelten Taint-Informationen zwischen Datenfluss basiertem und Daten- und Kontrollfluss basiertem Tainting existieren. Die Wahl der passenden Methode für eine Anwendung ist somit wichtig für den Nutzen von Dynamic Tainting.

Performance-Analysen zeigten eine beachtliche Verlängerung der Laufzeit und einen stark erhöhten Speicherverbrauch.

6 Zusammenfassung

Der Vorteil eines generellen Dynamic Tainting Frameworks liegt in den vielseitigen Einsatzmöglichkeiten. Es existieren sehr viele spezielle Ansätze, deren Einsatz auf einen Problembereich limitiert ist. Dynamic Tainting erfreut sich heutzutage auch in Bereichen außerhalb der Applikationssicherheit an Beliebtheit. Deswegen gewinnen generelle Ansätze an Bedeutung. Flexibilität stellt eine wichtige Anforderung für generelle Frameworks dar.

Dytan ist eine Implementierung des vorgestellten Frameworks und arbeitet auf x86-Binaries. Die Implementierung unterstützt Tainting auf Basis von Daten- und Kontrollfluss. Die Miteinbeziehung von Kontrollflüssen kann die Aussagekraft von dynamischen Taint-Analysen erhöhen, jedoch wird dann meist die Performance des Programms stark vermindert. Die Autoren aus [1] sind der Meinung, dass die Performance von Dytan noch Möglichkeiten zur Verbesserung bietet.

7 Referenzen

[1] Clause, J., Li, W., and Orso, A. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international Symposium on Software Testing and Analysis* (London, United Kingdom, July 09 - 12, 2007). ISSTA '07. ACM, New York, NY, 196-206.

[2] Lam, L. C. and Chiueh, T. 2006. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference* (December 11 - 15, 2006). ACSAC. IEEE Computer Society, Washington, DC, 463-472.

[3] Xu, W., Bhatkar, S., and Sekar, R. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Range of Attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Vancouver, B.C., Canada, July 31 - August 04, 2006). USENIX Association, Berkeley, CA, 9-9.

[4] Newsome, J. and Song, D. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium* (NDSS 2005).

[5] Haldar, V., Chandra, D. and Franz, M. 2005. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference* (December 05 - 09, 2005). ACSAC. IEEE Computer Society, Washington, DC, 303-311.

[6] Nguyen-Tuong, A., Guarnieri S., Greene D., Shirley J and Evans D. 2005. Automatically Hardening Web Applications Using Precise Tainting. In *20th International Information Security Conference* (IFIP 2005).

[7] Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M. and August, D. I. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture* (Portland, Oregon, December 04 - 08, 2004). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 243-254.

[8] Leek T., Baker G., Brown R., Zhivich M. and Lippmann, R. 2007. Coverage Maximization using Dynamic Taint Tracing. Technical Report TR-1112, MIT Lincoln Laboratory.