AUTOMATICALLY HARDENING WEB
APPLICATIONS USING PRECISE TAINTING
Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley,
David Evans

VERSUS

Precise Alias Analysis for Static Detection of Web Application Vulnerabilities
Nenad Jovanovic Christopher Kruegel Engin Kirda

Abstrakt

Meiste für Web geschriebene Applikationen beinhalten Schwachpunkte. Die zwei populärste sind "SQL Injection" und "cross-site scripting". Als Antwort dazu sind mehrere Tools entwickelt worden die solche Probleme erkennen bzw. mildern. Die erwarten aber Einmischung des Programmierers oder neigen zur Erkennung der "false positives".

Beide Arbeiten beschreiben verschiedene Betrachtungswaisen wie man an das Problem ran geht. Es ist eine Art Behandlung der unsicheren Daten, Im ersten Projekt werden gezielte Fragmente als unsicher betrachtet. Zweites Projekt konzentriert sich auf das Problem der Aliases.

1.Einführung

Meiste Webapplikationen sind eigentlich in sicherheitskritischem Bereich, aber eigentlich nur einige leisten sich eine umfassende Überprüfung der Sicherheitsaspekten, die mühsame Suche nach Feinheiten wird nicht durchgeführt. Daher auch bei Großprojekten wie hotmail, eBay oder Yahoo werden Sicherheitslücken gefunden.Einige Werkzeuge sind entwickelt worden, um Aspekte der Sicherheit, einschließlich statische Analyse des Quellcodes, und automatisierte Prüfwerkzeuge die vorgefertigte, absichtlich bösartige Daten liefern um Schwachpunkte zu erkennen . "Taint" Analyse kennzeichnet Daten, die von den unsicheren Quellen (einschließlich die von Benutzer eingegebenen) und folgt alle Daten, die durch diese Eingangswerte beeinflusst wird. Es wird ein Fehler gemeldet, wenn verdorbene Daten als Sicherheitkritischer Parameter übermittelt werden, wie ein Kommando, was zu einem exec Befehl geführt wird. "Taint" Analyse kann statisch oder dynamisch erfolgt werden. Projekt 1 geht dynamisch, Projekt 2 geht statisch an das Problem ran.

Damit eine Herangehensweise wirkungsvoll für die beträchtliche Mehrheit der Netzanwendungen wird, muss sie völlig automatisiert werden. Viele Leute errichten ihre Website ohne irgendein Verständnis der Sicherheitsproblemen. Z.B. versorgt "PHP und MySQL für Dummies" unerfahrene Programmierer mit dem Wissen, das sie eine Datenbankunterstützte Netzanwendung aufstellen müssen. Obgleich das Buch etwas Warnungen über Sicherheit einschließt (zum Beispiel, warnt auf einer Seite den Leser vor böswilligen Eingangsdaten sowie "<script>" Konstruktionen und zeigt korrektes Format), jedoch viele weitere Beispiele im Buch solche Lücken enthalten (z.B., Auflistungen 113 und 122 erlauben SQL injection, und Auflistung 124 erlaubt den crosssite scripting). Dieses ist für solche Büchern typisch.

Projekt 1 schlägt eine vollständig automatisierte Einheit für das Verhindern von von zwei wichtigen Kategorien der Sicherheitslücken vor: injection (einschließlich script und SQL injection) und den crosssite scripting(XSS). Die Lösung beinhaltet den StandardPHP Interpret mit einem geänderten Interpreten zu ersetzen, der genau die "befleckte" Daten aufspürt und auf Nutzung von solchen Daten in sicherheitskritischen Bereichen überprüft. Alles, was nötig wird, um daraus zu profitieren, ist, dass die User die geänderte Version von PHP verwenden. Der Hauptbeitrag der Arbeit ist die Entwicklung des exakten Tainting wo die Informationen auf einem feinen Niveau von Aufsplittung beibehalten und innen eine kontextsensitive Weise überprüften werden. Dieses ermöglicht, vollautomatisierte Abwehrmechanismen gegen command injection, einschließlich SQL injection und crosssite scripting Angriffen zu entwerfen und einzuführen.

Paper 2 ist ein Teil eines größeren Projektes : PIXY. Dieser Projekt ist ein Tool zur statischer Analyse des PHP Codes, was mit Hilfe control flow graph (JFlex / JCup) die taint, literal, alias und file inclusion analysis durchführt. In dem Paper wird die alias und file inclusion analysis besprochenen. Das ist relativ wichtiges Problem, denn die aliases in PHP sich von z.B. pointer in C unterscheiden und dia Autoren mussten ein neues Konzept durcharbeiten. Dazu das dynamische Inkludieren der Dateien bringt neuartige Probleme die in Compiler-Sprachen nicht gab. In der Arbeit ist auch nur Teil der allen Kombinationen beschieben.

2.Sicherheitslücken in Netzanwendungen

Das Bild stellt eine typische Netzanwendung dar. Für Klarheit konzentrieren wir auf die Netzanwendungen, die mit PHP eingeführt werden, das z.Z. eine der populärsten Sprachen für das Implementierung von Netzanwendungen ist (PHP wird an den ungefähr 1.3 Mio. IP Adressen, 18 Mio. Domains verwendet und wird auf 50% von Apache Servers installiert). Die meiste Verhaltensweise und die Eigenschaften sind für andere Websprachen ähnlich. Ein Klient schickt Daten zum Webserver in Form eines HTTP request (Schritt 1 in Tabelle 1). GET und POST sind die typischen "Abfragen". Der Abfrage kodiert/beinhaltet die Daten, die vom Benutzer in der HTTP header platziert werden, auffängt einschließlich die Dateinamen und Parameter, die im abgefragten URI eingeschlossen sind. Wenn das URI eine PHP Datei ist, lädt der HTTP Server die abgefragte Datei vom Dateisystem (Schritt 2) und lässt die im PHP Interpreten (Schritt 3) durchführen. Die Parameter sind zum PHP Code durch vorbestimmte globale variable Arrays sichtbar (einschließlich $_GET und $_POST). Der PHP Code kann diese Daten verwenden, um Befehle zu konstruieren, die zu den PHP Funktionen wie einer SQL-query geschickt werden, der zur Datenbank geschickt wird (Schritte 4 und 5) oder Anrufe PHP API zu den Funktionen bilden, die System API aufrufen, um den Zustand (Schritte 6 und 7) zu manipulieren. Der PHP Code produziert eine ausgehende Webseite, die auf den zurückgebrachten Resultaten basiert und bringt sie zum Klient zurück (Schritt 8). Es wird angenommen , dass ein Klient auf das web Server einwirken kann, nur indem er HTTP Abfragen zum HTTP Server schickt. Insbesondere ist die einzige Weise des Angreifers das System zu beeinflussen , einschließlich auf die Datenbank und Dateisystem einzuwirken, indem er passende Abfragen konstruiert. Die Angriffe werden in zwei allgemeine Kategorien geteilt: injection attacks - Abfragen zum Webserver, die seinen Zustand unerwünscht ändern oder vertrauliche Informationen aufdecken; output attacks - (z.B., cross-site scripting)versuchen,Anträge zum Webserver zu schicken, die es veranlassen, Antworten zu erzeugen, die böswilliges Verhalten für Klienten produzieren.

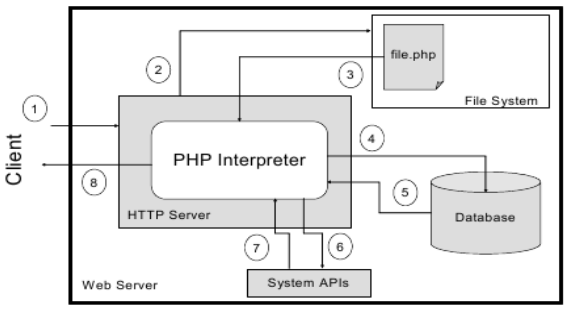

Figure 1. Typical web application architecture

2.1.Angriff Command injection

In einem command injection versucht ein Angreifer, vertrauliche Informationen zugänglich zu machen oder den Zustand der Applikation nachteilig zu ändern, indem er einen Eingang konstruiert, der dem Angreifer erlaubt, böswillige Steuerbefehle in die Netzanwendung einzusetzen. Bei der Architektur in Tabelle 1 könnte ein Angriff versuchen , PHP Code im PHP Interpreter, SQL Code im Datenbank oder native Code direkt auf der Maschine aufzurufen. Wir betrachten nur die ersten zwei Fälle. Sicherheitslücken in Webapplikationen sind weit populärer als die im Betriebsystemen, da es viel mehr unterschiedlichere Netzanwendungen als Betriebssysteme gibt. Dazu sind die meiste Webanwendungen weitaus weniger entwickelt.
PHP injection. In einem PHP injection Angriff versucht der Angreifer, PHP Code einzuschleusen, was von dem Interpreter ausgeführt wird. Wenn ein Angreifer beliebigen Code eingeben kann, kann der Angreifer alles tun was auch PHP kann und hat effektiv komplette Steuerung über dem Server. Das ist ein einfaches Beispiel einer PHP injection im phpGedView, einem online-Betrachtung System für Genealogie-Informationen. Das Angriff URL ist in der Form:

http://[target]/[...]/editconfig_gedcom.php?gedcom_config=../../../../../etc/passwd

Der verletzbare PHP Code verwendet den gedcom_config Wert als Dateiname: require($gedcom_config);. Die Semantik der Funktion require soll die Datei laden und entweder, sie als PHP Code zu deuten(wenn die PHP Tags gefunden werden) oder den Inhalt anzeigen. So entweicht dieser Code den Inhalt der Password-Datei aus. Missbrauch von require und ähnlichen Funktionen ist allgemein berichtet worden, obwohl, richtig konfiguriert, ist PHP für diesen Angriff undurchdringlich . Jedoch ist zusätzliche Verteidigung für hoch entwickelte injection - Angriffe wie das "Santy Worm" oder das phpMyAdmin attack erforderlich.

Sql injection. Dieser Angriff ist der populärste unter Angriffen auf Webanwendungen. Typische, einfache Form davon könnte folgend aussehen. Nehmen wir an, dass das folgende verwendet wird, um eine SQL Frage zu konstruieren, um Benutzer via Datenbank zu beglaubigen:

```
$cmd="SELECT Benutzer  FROM Benutzern WHERE Benutzer = '". $user. "' AND Kennwort = '". $pwd. "'";
```

Der Wert von $user kommt von $_post['user '], ein Wert, der vom Klienten mit der login-Formular geschickt wird. Ein böswilliger Klient kann den Wert eintragen: ' OR 1 = 1; --' (- - fängt ein Kommentar in SQL an, der zum Ende der Linie fortfährt). Die resultierende SQL Frage ist:

```
SELECT Benutzer FROM Benutzern WHERE Benutzer = ' ' OR 1 = 1; -- ' AND Kennwort = ' x '.
```

Der eingefügte Befehl schließt den Anführungsstrich und kommentiert Teil der Abfrage ab AND aus. Folglich wird die Abfrage als erfolgreich zurückgeliefert unabhängig vom Passwort. Das Hauptproblem hier ist, dass der einzelne Anführungsstrich, der vom Angreifer eingefügt wird, den geöffneten Anführungsstrich schließt, und der Rest der von Benutzer erstellter Zeichenkette wird zur Datenbank als Teil des SQL Befehls weitergehen. Dieser Angriff würde durch PHP Installationen vereitelt, die die voreingestellt magic quotes verwenden. Wenn es erlaubt ist, säubern magic quotes automatisch die Eingangsdaten, indem sie einen backslash allen Zeichenketten hinzufügen, die über Formulare oder cookies kommen. Jedoch genügt es nicht für Angriffe, die die Anführungsstriche nicht verwenden. Eine Lösung, zum der SQL injection zu verhindern sollen vorbereitete Verfahren sein. Ein vorbereitetes Verfahren ist eine Frage Zeichenkette mit Platzhaltern für Variablen, die nachher zur Aussage eingefügt werden und nach richtigkeit überprüft werden. Jedoch hängt dieses von den Programmierern ab, ob die die Entwicklungspraxis ändern und schon existierendes Code ersetzen. Dynamisches Erzeugung der Abfragen wird weiter in der Zukunft überwiegen.

2.2.Output Angriffe

Output Angriffe schicken eine Abfrage zu einer Netzanwendung, was verursacht, dass von dem Angreifer erwartete Ausgangsseiten produzieren werden,die irgendein böswilliges Ziel erzielen. Die gefährlichste Art des output Angriffs ist ein cross-site scripting Angriff, in dem das Webserver eine Ausgangseite produziert, die den Code enthält, der durch den Angreifer erzeugt wird. Der Code kann die cookies des Opfers stehlen, oder Sicherungdaten, die das Opfer ohne Verdacht auf der Website eingibt. Dieses ist bei phishing Angriffen besonders wirkungsvoll, in denen der Angreifer möglichen Opfern die email schickt, die sie Opfer überzeugen, ein URL zu besuchen. Das URL kann eine vertraute Domain sein, aber wegen einer cross-site scripting kann der Angreifer Parameter zum URL konstruieren, die der eigentlich sicherer Webseite veranlassen, eine Seite herzustellen, die ein Formular enthält, die Daten zurück zu dem Angreifer sendet. Z.B. konstruiert der Angreifer eine Verbindung so:

```
<a href='http://bad.com/go.php?val=<script src="http://bad.com/attack.js"></script>'>
```

Wenn die Implementierung von go.php der val Parameter im erzeugter Webseite verwendet(zum Beispiel, bei print"results for: ". $_get['val '];), der böswillige Script erscheint auf der resultierenden Seite. Ein kleverer Angreifer kann character encodings verwenden, um den böswilligen Script für das Opfer sinnlos aussehen zu lassen, das das URL vor Öffnung kontrolliert. Vor fünf Jahren, beschrieb CERT advisory 2000-02 das Problem cross-site scripting und riet Benutzern, Scriptsprachen zu sperren, und Website-Entwickler die Webseite-output zu kontrolieren. Dennoch cross-site scripting bleibt ein ernstes Problem heute. Weit zu viel Funktionalität des Netzes hängt von scripting Sprachen ab, so die meisten Benutzer werden es nie sperren. Sogar Sicherheitsbewusste Netzentwickler erstellen häufig Websiten, die cross-site Problem. Wie mit SQL injection, entdeckte Probleme die schnell fixiert werden, können nicht den breiten Angriffsspektrum standhalten. Folglich konzentrieren man sich bei beiden Projekten auf völlig automatisierte Lösungen.

4. PROJEKT 1 : AUTOMATIC WEB HARDENING

Diese Konstruktion basiert auf dem Beibehalten der exakten Informationen, welche Daten durch die Verarbeitung eines Antrags verdorben werden und Prüfung, ob Eingangsdaten, die zu einem externen Befehl oder zu einem Ausgang zu einer Webseite geschickt werden, nur sicheren Inhalt enthalten. Unsere automatisierte Lösung verhindert eine große Gruppe der Attacken ohne direkter Bemühung der Programmierer. Die einzige Änderung zu der Standardnetzarchitektur in Tabelle 1 ist, dass wir den Standard-PHP Interpreter mit einem geänderten ersetzen, der kennzeichnet, welchen Daten aus unsicheren Quellen kommt und genau schaut, wie diese Daten durch PHP Interpreter (Abschnitt 4.1) sich verbreiten, prüft, ob Parameter zu den Befehlen nicht den gefährlichen Inhalt enthalten, der vom Benutzereingang (Abschnitt 4.2) abgeleitet wird, und sicherstellt, dass erzeugte Webseiten kein Code enthalten, was aufgrund unsicherer Eingangsdaten erstellt wird (Abschnitt 4.3).

4.1.Das Verfolgen exakter tainting-Informationen

Die Eingangsdaten aus unsicherer Quellen werden kennen gezeichnet, einschließlich die Daten, die von den Klient kommen, als verdorben. Wir änderten die Implementierung des Datentyps string im PHP Interpreter, um die "Verdorbenheit" der einzelnen Buchstaben darzustellen. Wir verbreiten dann diese Informationen durch Funktionsaufrufe, Anweisungen und Aufbau am Aufsplittung eines einzelnen Buchstabens, folglich - precise tainting . Die Anwendung des precise tainting ermöglicht die Verhinderung der injection-Angriffe und Filtern des Ausgangsdaten wegen XSS Angriffe. Wenn eine Funktion eine verdorbene Variable in einer gefährlichen Weise verwendet, können wir den Anruf der Funktion ablehnen(wie mit SQL Abfragen oder PHP Systemfunktionen getan wird) oder die Variablenwerte sanieren (wie für das Verhindern cross-site Angriffen getan wird). Netzanwendung Entwickler denken daran häufig, sich Eingangsdaten von GET und POST zu sanieren , aber vergessen, die andere Variablen zu überprüfen, die von den Klienten manipuliert werden können. Unsere Auffassung stellt sicher, dass all diese externe Variablen, z.B. hidden-Formularvariablen, cookies und HTTP headers, gekennzeichnet sind als verdorben. Wir verfolgen auch tainting-Informationen für sesion-Variablen und Datenbankresultate.

4.1.1Taint strings
Für jede PHP Zeichenkette, wir folgen tainting-Informationen für einzelne Buchstaben. Betrachten Sie das folgende Codefragment, wohin der Teil der Zeichenkette $x von einem Netzformular und anderseits von einem cookie kommt:

```
$x = "hallo". $_GET['name1 ' ]. ". Ich bin ". $_COOKIE['name2 ' ];
```

Die Werte von $_GET['name1 '] und von $_COOKIE['name2 '] werden völlig verdorben (wir nehmen an, dass sie Alice und Bob sind). Nach der Zussamensetzung sind die Werte von $x und seine Markierungen (unterstrichen):

Hallo Alice. Ich bin Bob.

4.1.2 Funktionen
Es werden tainting-Informationen über Funktionsanrufe verfolgt, insbesondere Funktionen, die Zeichenketten manipulieren und zurück liefern. Der allgemeine Algorithmus soll die Zeichenketten kennzeichnen, die von der Funktion zurückgeliefert werden, wie verdorben, wenn irgendwelche der Eingang Argumente verdorben sind. Wann immer durchführbar, wird die Semantik von Funktionen genau verfolgt. Z.B. betrachten Sie die substring Funktion, in der tainting-Markierungen des Resultats des substr Anrufs von Teilen der Zeichenkette abhängen, die sie aufrufen:

```
substr("precise taint me", 2, 10); //ecise tai
```

4.1.3.Datenbankwerte und session-Variablen
Datenbanken stellen einen anderen möglichen Schauplatz für Angreifer zur Verfügung, um böswillige Werte einzusetzen. Wir behandeln Zeichenketten, die von den Datenbankfragen zurückgebracht werden wie unsicher und kennzeichnen sie, wie verdorben. Während diese Auffassung übermäßig einschränkend aussehen kann, in der Richtung, dass legitimer Gebrauch verhindert werden kann, zeigen wir in Abschnitt 4.3, wie das precise tainting und unsere Auffassung der Überprüfung auf cross-site scripting dieses mögliche Problem abschwächt. Weiter wenn die Datenbank auf irgendeine andere weise kompromitiert wird, ist der Angreifer noch nicht imstande, die verglichene Datenbank zu benutzen, um einen cross-site scripting Angriff zu konstruieren.
Die statische Natur von HTTP erfordert vom Entwickler, Zustand der Anwendung über Abfragen zu verfolgen. Jedoch würde das Enthüllen von Klient-session-Variablen den Angreifern erlauben, Anwendungen zu manipulieren. Gut entworfene Netzanwendungen halten session-Variablen nur auf dem Server und benutzen eine session-id, um mit Klienten zu kommunizieren. Wir änderten PHP, um tainting-Informationen der session-Variablen zu speichern.

4.2.Verhindern der command injection

Die tainting-Informationen werden verwendet, um festzustellen, ob Anrufe zu den Sicherheit-kritischen Funktionen sicher sind oder nicht. Um command injection Angriff zu verhindern, prüfen wir ob die verdorbenen Informationen, die zu einem Befehl geführt werden sicher sind. Die tatsächliche Überprüfung hängt vom Befehl ab, und ist exakt genug, alle command injection bei typischen Netzanwendungen zu verhindern.

4.2.1 Die PHP injection
Um PHP injection Angriffe zu verhindern verweigert man Anrufe der möglicherweise gefährlichen Funktionen an, wenn irgendein ihrer Argumente verdorben wird. Die Liste der Funktionen, die überprüft werden, ist denen ausgeschalteten im Perl und Ruby tainting mode ähnlich und besteht aus Funktionen die Zeichenketten behandeln, den Zustand des Systems manipulieren wie Systemanrufe oder I/O Anrufe.

4.2.2 Die SQL injection
Das Verhindern der SQL Einspritzungen erfordert das Nutztung exakten tainting-Informationen. Vor Absenden eines Befehls zur Datenbank, z.B. mysql_query, lassen wir den folgenden Algorithmus zur Überprüfung auf injections laufen:

1.Teile die Abfrage auf Tokens; behalte die taint-Markierung mit den Tokens.
2. Skane jedes Zeichen für Bezeichner und Operatorsymbole ab (ignoriere Symbole d.h. Zeichenketten, Zahlen, Boolesche Werte).
3. Erkenne eine injection, wenn ein Operatorsymbol als verdorben gekennzeichnet wird. Operatorsymbole sind ,()[].;:+-*/\%^ <>=~!?@#&|`
4. Erkenne eine injection, wenn ein Bezeichner verdorben und ein Schlüsselwort ist . Beispielschlüsselwörter UNION, DROP, WHERE, OR, AND.
Beispiels von Abschnitt 2.1: $cmd="SELECT Benutzer vFROM Benutzern WHERE Benutzer = '". $user. "' AND Kennwort = '". $password. "'";
Die resultierende Abfrage (mit $user gesetzt zu ' OR 1 = 1; -- ') wird wie folgt verdorben:
SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password= ' x '.
Wir erkennen eine injection bei OR, was verdorben *und* ein Schlüsselwort ist.

4.3 Das Verhindern des cross-site scripting
Diese Auffassung an das Verhindern des cross-site scripting beruht auf der Überprüfung des erzeugten Ausganges. Jeder möglicherweise gefährliche Inhalt in erzeugten HTML Seiten darf keine verdorbene Daten enthalten. Wir ändern die PHP Ausgangsfunktionen (print, echo, printf und andere Druckfunktionen) mit Funktionen, die auf tainting die Ausgangsdaten überprüfen. Der ersetzte Funktionen betrachten sicheren Text normalerweise, betrachten den Zustand der Ausgangsdaten als unsicher. Für ein Beispiel betrachten Sie eine Anwendung, die einen Index öffnet und dann verdorbenen Ausgang druckt:

```
print "<script>document.write ($user)</script>";
```

Ein Angreifer kann Javascriptcode einsetzten, indem er den Wert von $user auf einen Wert einstellt, der die Klammern schließt und beliebigen Code durchführt: "me");alert("yo". Merken Sie, dass der öffnende tag über mehrfache Druckbefehle geteilt werden könnte. Folglich müssen unsere geänderten Ausgangsfunktionen die geöffneten und teilweise geöffneten tags im Ausgang verfolgen. Wir brauchen nicht, das Ausgangs-HTML vollständig zu analysieren (es ist ratsam so zu tun, da viele Netzanwendungen grammatisch unkorrektes HTML erzeugen).
Die Überprüfung des Ausganges anstelle vom Eingang vermeidet viele der allgemeinen Probleme mit ad hoc Filtern. Die Überprüfung beinhaltet sicheren Inhalt auf die whitelisting zu setzten, während blacklist versuchen cross-site scripting Angriffe zu verhindern, indem es bekannte gefährliche tags, wie <script> und <object> kennzeichnet. Das letzte kann nicht die injection verhindern, die andere tags ermöglichen.
Z.B. kann ein Index in den anscheinend harmlosen <b> (fett) tag mit Parametern wie onmouseover inieziert werden.
Diese Verteidigung nimmt Nutzten aus der exakten tainting-Informationen, zu Identifizierung der Webseiten die aus unsicheren Quellen entstanden sind. Jeder möglicher verdorbene Text, der gefährlich sein könnte, wird entweder vom Ausgang entfernt oder geändert, um zu verhindern,dass er interpretiert wird (zum Beispiel, ersetzen < in unbekannten tags mit <). Die konservativen Annahmen bedeuten, dass irgendein sicherer Inhalt unbeabsichtigt unterdrückt werden kann; jedoch wegen der exakten tainting-Informationen, wird dieses begrenzt zur Inhalten die von unsicheren Quellen erzeugt werden.

4.4. Projekt 1 : Zussamenfassung

Es wurde eine völlig automatisierte, end-to-end Auffassung für das Sichern von Netzanwendungen beschrieben. Indem das präzise tainting in einer Weise genutzt wird, die Nutzen der Programmiersprachen-Semantik und Durchführung der Kontext-abhängigen Überprüfung bezieht, ist man, eine große Kategorie Angriffen zu verhindern, ohne irgendeine Bemühung zu vom Programmierer zu erfordern. Messungen zeigen an, dass die zusätzliche Kosten die genommen werden, indem man das geänderte Interpreter verwendet, kleiner als 10% sind.

Wirkungsvolle Lösungen für schützende Netzanwendungen müssen die Notwendigkeit an der Präzision mit der begrenzten Zeit ausgleichen. Völlig automatisierte Lösungen, wie die, die in dem ersten Paper beschrieben wird, liefern einen wichtigen Punkt in diesem Designraum.

5.PROJEKT 2 : PIXY, ALIAS ANALYSIS

Problematik des Aliases ist bei PHP sicherheitskritisch, denn bildet die eine Quelle der Lücken die diecht durch Tainting zu entdecken sind wenn man es nicht beachtet. Es folgt quasi "stille" Übernahme der befleckten Daten :

```
1: $b = 'nice'; // $b: untainted
2: $a =& $b;    // $a and $b: untainted
3: $a = $evil;  // $a and $b: tainted
4: echo $b;     // XSS vulnerability
```

5.1 Untypisches Verhalten

```
1: $x1 = 1;
2: $x2 = 2;
3: a(&$x1);
4: echo $x1;   // $x1 ist weiter '1'
5:
6: function a(&$p) {
7:   $p =& $GLOBALS['x2']; //hier wird nur $p rereferenziert, nicht $x1
8: }
```

In diesem Beispiel sieht man : $p wird zum Alias zur $x1 wegen Funktionsaufruv "by reference". Dann wird in Zeile 7 Variable $p zu Alias der Variable $x2. ABER : dadurch ändert sich die Verbindung der $p, und $x1 bleibt unberührt auch wenn sie mal alias zu $p war.
Eigentlich alle Variablen sind aliases zur Skalaren, mancher Skalar hat eben nur ein Alias.

5.2 Analyse des Ablaufes innerhalb einer Prozedur

```
1: skip;       // u{} a{}
2: $a =& $b;   // u{(a,b)} a{}
3: if (...) {
4:   $c =& $d; // u{(a,b) (c,d)} a{}
5:   $e =& $d; // u{(a,b) (c,d,e)} a{}
6: }
7: skip;       // u{(a,b)} a{(c,d) (c,e) (d,e)}
```

Es werden 2 Wege analysiert:
must Aliases (u{}) - Aliasen die in einem Verlaufpunkt auf jedem Fall vorkommen, unabhängig von z.B. Eingabedaten
may Aliases (a{}) - Aliasen die je nach Verlauf (if oder while Anweisungen) vorkommen können aber bei bestimmten Verlauf doch nicht stattfinden können
Das erlaubt präzise Einstellung wie wir vorgehen wollen : konservativ oder aggressiv.

5.3 Analyse des Ablaufes zwischen Prozeduren
Hier zu beachten sind 2 Typen der Variablen
- lokale, wo auch die Funktionsparameter dazu gehören
- globale, über "global $x" und $GLOBALS["x"] in Funktionen zugreifbar

Das größte Problem, Rekursion, was zur unendlichen Verläufen führen kann, wird damit gelöst, dass man nur globale Variablen und lokale Inkarnationen (Variablen die immer wieder bei der rekursiven Aufrufen erscheinen) betrachtet.

5.3.1 Aliases zwischen globalen Variablen

```
01: skip;   // u{} a{}
02: a();
03: skip;   // u{(m.x1, m.x2, m.x3)} a{}
04:
05: function a() { // u{} a{}
06:   $a1 =& $a2;    // u{(a.a1,a.a2)} a{}
07:   $GLOBALS['x1'] =& $GLOBALS['x2'];
08:   skip;       // u{(a.a1,a.a2) (m.x1, m.x2)} a{}
09:   b();
10:   skip;       // u{(a.a1,a.a2)
11:               //   (m.x1, m.x2, m.x3)} a{}
12: }
13:
14: function b() { // u{(m.x1, m.x2)} a{}
15:   $GLOBALS['x3'] =& $GLOBALS['x1'];
16:   skip; // u{(m.x1, m.x2, m.x3)} a{}
17: }
```

Der Tracking wird dadurch geführt dass man die lokalen Variablen mit Präfix der Funktion bestückt (a.a1) , die globale mit m (main) - m.x1
wichtig : welche globale Variablen haben bei Aufruf einer Funktion ein alias und welche alias Informationen waren bei dem Aufruf geändert.

5.3.2 Aliases zwischen globalen Variablen und lokalen Variablen der aufgerufener Funktion

```
01: a();
02: skip;   // u{(m.x1, m.x2)} a{}
03:
04: function a() {  // u{(m.x1, a.x1_gs)
05:                 //   (m.x2, a.x2_gs)} a{}
06:   $a1 =& $GLOBALS['x1'];
07:   skip; // u{(m.x1, a.x1_gs, a.a1)
08:         //   (m.x2, a.x2_gs)} a{}
09:   b();//nach $ diesem aufruf ist $a1 kein
alias zur $x1 mehr
10:   skip; // u{(m.x1, m.x2, a.x2_gs)
11:         //   (a.a1, a.x1_gs)} a{}
12: }
13:
14: function b() {   // u{(m.x1, b.x1_gs)
15:                  //   (m.x2, b.x2_gs)} a{}
16:
17:   $GLOBALS['x1'] =& $GLOBALS['x2'];
18:
19:   skip;   // u{(m.x2, b.x2_gs, m.x1)} a{}

20: }
```

Eigentlich kann man die lokale aliases nicht in aufgerufener Funktion ändern, es sei denn :

- lokale Variable ist ein alias zur einer globalen Variable und
  - es kommt ein alias auf die globale Variable und somit auch auf die lokale
  - die globale variable wird zu was anderes umgeleitet
- lokale Variable ist ein alias zum fomalen Parameter und
  - eine globale Variable wird zum alias des Parameters und somit zu der lokalen Variable

Es wird mit so genannter "shadow variable" gehandhabt. Bei einem Funktionsaufruf jede Variable bekommt ein "Schatten" - eine lokale Variable, die die gleichen Aliases hat. Beim Rückkehr aus der Funktion sieht man was sich geändert hat. Man unterscheidet:

- _fs formal shadow
- _gs global shadow

5.4 Begrenzungen

Das Projekt unterstützt keine OOP. Es ist stark für PHP 4 konzipiert. Des weiteren es wird nicht an Fälle wie Aliases zwischen Tabellen und Tabellenelementen ran gegangen

5.5 Resultate

Program	Entry Files	LOC	Time (sec/File)	Vulnerabilities	False Positives	BugTraq ID
DCP Portal 6.1.1	22	61 617	6.0	61	35	427175
MyBloggie 2.1.3beta	6	20 326	58.3	14	5	427182
TxtForum 1.0.4-dev	15	4 398	1.3	31	17	427186, 427188
Totals	**43**	**86 341**	**11.7**	**106**	**57**	

Es scheint dass es viele False Positives gibts (ca. 50%) , jedoch sind es meistens ähnliche, oder sicherheits-irrelevante Fälle. Zu den wichtigsten Gruppen gehören:
- predefinierte Variablen wie $ SERVER['PHP_SELF'] die nicht beeinflussbar sind
- reguläre Ausdrücke, die eigentlich " sanitized" sind
- Unmögliche oder praktisch sichere Programmabläufe

5.5 Auflösung der rekursiven include-Aufrufen
PHP Projekt sind meistens in vielen Dateien verteilt, deren Namen auch dynamisch erstellt werden können
Herangehensweise in 2 Schritten

- auflösen der Dateien deren Name direkt bekannt ist
- suchen aufgrund der literarer Analyse aus P.1 nach neuer include Dateien
- die 2 Punkte rekursiv

Dabei werden "echte" include-Rekursionen erkannt (Datei ruft sich selbst mit gleichen Parameter auf) und mit no-op ersetzt.

Die Efektivität:

Program	Iterations	Resolved Literal Includes	Resolved Non-Literal Includes	Unresolved Includes
DCP Portal 6.1.1	3.9	0.9	5.8	1.9
MyBloggie 2.1.3beta	3	6.5	12.3	0
TxtForum 1.0.4-dev	1.5	1.8	2.6	1.7

5.6 Fallstudie

```
01: if (...) {
02:   multi_tb($post_urls, ...);
03:   $tbstatus = $tbstatus . $tbreply;
04: } else {
05:   $tbstatus ="";
```

```
06: }
07:
08: message(..., $tbstatus);
09:
10: function message(..., $message ) {
11:   echo $message;
12: }
13:
14: function multi_tb($post_urls, ...) {
15:   global $tbreply;
16:   $tb_urls = split('( )+', $post_urls, 10);
17:   foreach($tb_urls as $tb_url) {
18:     $tbreply .= $tb_url;
19:   }
20: }
```

Es ist ein Vereinfachtes Beispiel aus MyBloggie.
Kaum manuell zu finden :
Variable $post_urls ist tainted aber landet in der echo-Ausgabe.

5.7 Projekt 2 : Zussamenfassung

Die neue Herangehensweise der Autoren ermöglicht erst weitestgehend flächendeckend tainting bei PHP mit Beachtung der typischen Merkmale der Sprache. Ein Tool was sehr hilfreich beim Audit-arbeiten erscheint.

6 VERGLEICH / ZUSSAMENFASSUNG

Beide Projekte Befassen sich mit gleichem Gebiet : Sicherheitslücken bei PHP Scripten. Beide benutzten auch die Methode des Taintings. Jedoch technische Herangehensweise ist anders : Projekt 1 arbeitet dynamisch durch den Austausch den PHP - Interpreters, Projekt 2 arbeitet statisch mit Hilfe Java-Tools. Projekt 1 im Gegenteil zu dem 2. arbeitet mit der Tainting ganz präzise : auf der Buchstaben-Ebene (und nicht kompletter Variablen-Ebene).

Durch die Verschiedenheit ergänzen sich die 2 Tools ganz gut und liefern beinahe flächendeckende Lösung für die Sicherheit.