オープンソース・ソフトウェアの
セキュリティ確保に関する調査報告書

第Ⅱ部

オープンソース・ソフトウェアの効率的な検査技術の調査

情報処理振興事業協会

セキュリティセンター

目　次

図 目 次

表 目 次

1. 概要

1.1. 調査概要

オープンソース・ソフトウェアでは一般的な商用ソフトと違い、ソースコードを利用することが可能である。そこで、利用者側で、ソースコードを利用して、ソフトウェアに含まれているバグや脆弱性の検査を行うことが可能である。この検査の結果に従い、脆弱性の評価を行い、システムのセキュリティ上のリスクの把握を行うことは、利用者にとって重要である。現在、ソースコードの検査を効率的に行うための幾つかのツールがオープンソース・ソフトウェアとして公開されている。本調査は、オープンソース・ソフトウェアである代表的なソースコード検査ツールについて、どのように脆弱性がチェックされ、どのような検査報告が行われるか等について調査するものである。

1.2. 調査内容

本調査では、ソースコードの検査技術について以下の項目を明らかにするために調査を行った。

- ソースコードの検査技術の洗い出しと詳細分析
- 代表的なソースコードの検査技術の比較

1.3. 調査方法

1.2で述べた各調査項目に対して、以下の方法で調査を行った。なお、調査については、インターネット上で収集可能な公開情報をベースとし、実際に該当する技術を利用・検証した。

1.3.1. ソースコード検査の技術の洗い出しと詳細分析

ソースコード検査の技術については比較可能なように、以下の項目で分析を行った。

A) 概要
B) 検出可能な脆弱性
C) 機能詳細
D) 開発体制
E) 現在のバージョンと対応プラットフォーム
F) 利点と欠点
G) 対応言語

1.3.2. ソースコード検査技術の比較

ソースコードの検査技術を比較するために、CERT/CC 等でこれまで報告されてきている脆弱性を持つアプリケーションをターゲットとして、実際に検査ツールを利用して、ソースコード検査を行う。ただし、アプリケーションの全ソースコードの検査を行うと、検査能力の比較が難しくなるため(誤検出等が発生するため正しく脆弱性が検出されたか評価できないため)、アプリケーションに含まれる特定の脆弱性についてのみ検査を行う。今回対象とするアプリケーションは以下の通りである。

#	アプリケーション	情報源	脆弱性
1	bind8.1	CA-98.05	バッファーオーバーフローバグ
2	qpopper2.4	CA-98.08	バッファーオーバーフローバグ
3	elm2.5.0		バッファーオーバーフローバグ
4	imapd3.6	CA-98.09	バッファーオーバーフローバグ
5	apache1.1.3		/tmp へのシンボリックリンクバグ
6	wu-ftpd2.6.0	CA-00.13	フォーマット・ストリング・バグ

表1. 検証用のプログラム

検証においては以下の項目で比較を行った。

(1) 検査結果
(2) パフォーマンス
(3) 特定の脆弱性の検出に対する設定の容易さ

1.4. ソースコード検査の技術の分類と調査対象

効率的なソースコードの検査技術に関して本調査では、当該技術を以下の 2 つに分類し調査を行った。

(1) パターンマッチング技術

脆弱性に関するデータベースを元に、ソースコード中に脆弱性を持つ関数や変数が使用されているかどうかを検査する技術である。検査は、データベースに登録された脆弱性情報とソースコードのパターンマッチングによって行われる。

(2) 構文解析技術

ソースコードをそのまま検査するのではなく、構文解析を行い抽象化した構文木に変換し、検査を行う技術である。従来の構文解析技術に、セキュリティ検査の為の機能を付加したものである。セキュリティ検査のための付加的な注釈を使用し、このデータ遷移の安全性まで検査可能である。

上記の分類に従って、下記の 5 つのツールについての調査を行った。

#	分類	調査対象
1	パターンマッチング技術	RATS
2		ITS4
3		Flawfinder
4	構文解析技術	Splint
5		Cqual

表2.　調査対象一覧

2. 分類に基づく各セキュアな実行コードの生成、実行環境技術の分析

前述、1.4ソースコード検査技術の分類方法の項で説明した分類方法に基づき、それぞれの検査技術に対し、詳細分析を行った。

2.1. パターンマッチング技術

パターンマッチング技術とは、プログラマが grep コマンドを使って、脆弱性のある関数の場所を特定する作業を模倣した技術である。プログラマが経験・知識として蓄えている脆弱性に関わる情報をデータベース(脆弱性データベース)に記述し、その脆弱性データベースとソースコード中の関数で一致する個所が現われれば、脆弱性の可能性のある個所として判断する。多くのパターンマッチング技術を採用したツールでは、ある程度の誤検出は許容する考え方に基づいており、少しでも脆弱性の可能性がある場合はユーザに警告を出力する。本章では、パターンマッチング技術のツールとして、RATS、ITS4、Flawfinder の 3 つのツールについて説明する。

2.1.1. RATS に関する調査

A） 概要

RATS（Rough Auditing Tool for Security）は、米国セキュアソフトウェア社によって開発が行われているオープンソース・ソフトウェアである。ライセンス形態として GPL[1] が採用されており、自由に利用・再配布が可能である。 RATS を用いることで、C、C++、Perl、PHP、Python といったさまざまな言語の検査を行うことが可能である。検査可能な脆弱性としては、バッファーオーバーフローやレースコンディション等の脆弱性が挙げられる。検査は、内部に保有する脆弱性データベースに基づいて行われる。脆弱性データベースには約 500 個の脆弱性に関する情報が格納されており、脆弱性データベースをユーザが独自で作成することも可能である。

B） 検出可能な脆弱性

RATS において検査の対象となるのは、主に以下の 4 種類の関数群である。

(1)バッファーオーバーフローを引き起こす可能性の高い関数

例：gets, strcpy, strcat, printf, sprintf, scanf, sscanf, fscanf, vfscanf, vsprintf, vscanf, vsscanf, streadd, strecpy, strtrns, getchar, fgetc, fgets, fprintf

(2)レースコンディションを引き起こす可能性の高い関数

例：access, creat, fopen, lstat, stat, open, mkdir, rmdir, mktemp, opendir

[1] 本報告書では特に断らない限り、GNU General Public License Version2 を GPL と記述する。

(3)UNIX 用システムコール関数

例：system, chroot, popen, getenv, syslog, getopt, getpass, getlogin, ttyname

(4)乱数生成に関する関数

例：drand48, erand48, initstate, random, seed48, setstate, srand, srandom

C） 機能詳細

RATS の機能を次の 6 つの観点で解説する。

・データ入力
・ソースコードの検査
・脆弱性データベースの拡張
・検査の制御
・結果の出力
・開発環境との連携

データ入力

ユーザが RATS に引き渡すことができるデータは、検査対象となるソースコードと脆弱性データベースの二つである。

①検査対象となるソースコードの読み込み

ユーザが検査対象として指定したファイル名及びディレクトリ名を元に、ソースコードの読み込みを行う。検査対象としてディレクトリ名を指定した場合には、サブディレクトリ以下を再帰的に検査を行う。

②脆弱性データベースの指定

ユーザが独自に作成を行った脆弱性データベースを指定することが可能である。脆弱性データベースを指定した場合も標準で用意されている脆弱性データベースが読み込まれる。標準の脆弱性データベースの読み込まずに検査の実施をする機能も存在する。

③ソースコードの記述言語の識別

RATS には、ソースコードの記述言語の識別機能がある。検査対象となるファイルの読み込み時に、ファイル名の拡張子から記述言語を識別している。検査対象としてディレクトリ名が指定された場合、予め登録された拡張子 (.c、.pl、.py、.php 等) のファイルのみ読み込まれる。

ソースコード検査

検査は､脆弱性データベースとソースコード中で使用されている関数のパターンマッチングによって行われる。検査の精度の向上・効率化のために次のような機能が用意されている。

①データベースを用いたパターンマッチング

RATS は、起動時に脆弱性データベースの読み込みを行う。脆弱性データベースの情報は、読み込まれた順にメモリ領域にスタックされる。スタック情報に格納された情報と、ソースコード中の記述の比較によって検査を行う。比較はソースコードの先頭から行われる。比較を行う情報は、脆弱性データベースに記述された関数名とソースコード中で宣言・使用された関数名である。脆弱性データベースの情報とソースコードの記述の比較を効率的に行うために、脆弱性データベースの読み込み時に関数名情報のハッシュを行っている。この結果、関数名の比較を効率的に行うことが可能である。以下にパターンマッチングの概要をまとめた図を示す。

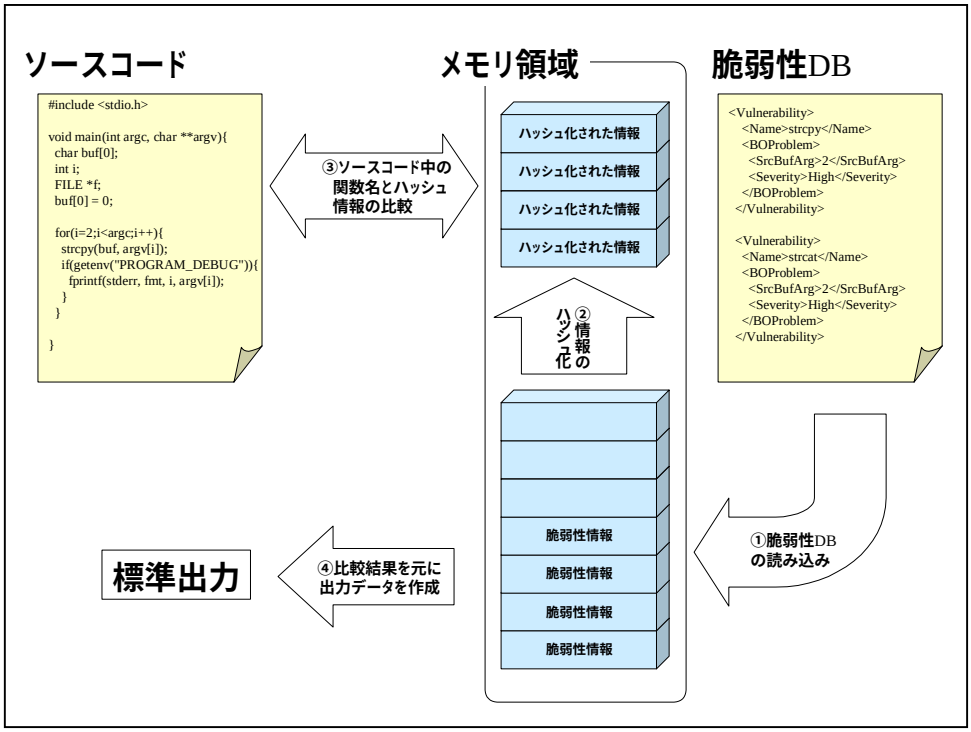

図1.　RATS の動作

②構文解析による変数名と関数名の識別

脆弱性データベースに登録された関数名と同じ名前の変数を宣言した場合、RATS は検出を行わない。これは、内部で簡単な構文解析を行っているためである。この

構文解析によって、引数の個数から安全であると判断される場合の誤検出は少なくなる。構文解析は以下のようにして行っている。

ⅰ：脆弱性データベースに関数名に伴って、引数の個数の指定を行う。

ⅱ：RATS は読み込んだソースコード中に、名前が合致する個所を見つけ出す。

ⅲ：合致した個所に“（”と“）”が記述されているかを検査し、且つ脆弱性データベースに記述された引数の個数について検査を行う。

解析の判断を以下のようなコードに対して、確認をする。

```
1  int printf;
2  int printf();
3  int printf(test1);
4  int printf(test1, test2);
```

上記では、printf という文字列を 4 種類記述している。この記述が検出されるかの確認を行う。以下は標準の脆弱性データベースにおける printf 関数の記述である。

```
<Vulnerability>
    <Name>printf</Name>
    <FSProblem>
      <Arg>1</Arg>                    ←引数の個数を指定
      <Severity>High</Severity>
    </FSProblem>
  </Vulnerability>
```

上記の脆弱性データベースでは、printf 関数には一つ以上の引数が引き渡されることが記述されている。上記のデータベースを用いて検査を行った結果は以下である。

コード	検出の可否
1 行目	×
2 行目	×
3 行目	○
4 行目	○

1 行目における宣言では、“（）”が使われていないために、関数として判断されない。2 行目における宣言では、“（）”が使われているために関数として判断されるが、引数が一つも渡されていない為に、検出は行われない。3 行目では、“（）”が使われおり、引数として文字列が記述されていることから、printf 関数が使用されていると検出される。4 行目では 3 行目と同様に複数個引数が渡されていることから、printf 関数が使用されていると検出される。

③コメント文・マクロ文の検査対象からの除外

コメント文に記述された関数名については、パターンマッチングは行われない。コメント文に対しては、後述の検査の省略にあるように、RATS 特有のコマンドを挿入可能である。従って、コメント文はソースコードの検査対象としては扱われないが、検査の効率化の対象としては扱われる。

マクロ文に関しての検査も行われない。マクロ文で脆弱性の可能性の高い関数名が変換されている場合には注意が必要である。マクロ文で脆弱性の可能性の高い関数名が変換されている場合には、変換後の記述形式を脆弱性データベースに登録することで、検査可能である。

④サイズの指定されたバッファの宣言の検出

RATS では、他のパターンマッチング型のツール（ITS4）と異なり、サイズの指定されたバッファの宣言が検出可能である。サイズの指定されたバッファの宣言の検出は、脆弱性データベースには記述されておらず、RATS の内部エンジンが自動的に検査を行う。

```
1   strcpy(buf, “test¥n”);
2   strcpy(buf, argv[1]);
```

⑤安全な引数の判別

関数の引数として渡されたデータは、プログラムのバグにはなるが脆弱性の要因とならないデータ（プログラム内部で宣言される文字列）と、脆弱性の要因となるデータ（ユーザが入力値を決定可能）に分類可能である。脆弱性の要因とならないデータを安全な引数と呼ぶ。RATS では、幾つかの関数（strcpy 等）の検査において、安全な引数の引渡しが行われる場合には、脆弱性の可能性のある関数を使用しているが安全な使用をしているとして脆弱性の検出を行わない機能がある。例として、以下のような記述に対して検査を行う。

```
1   strcpy(buf, “test¥n”);
2   strcpy(buf, argv[1]);
```

2 行目において、ユーザが buf のサイズより大きい値を入力することができるため脆弱性の可能性が検出される。一方で 1 行目においては、内部で定義された文字列であることから脆弱性の可能性は検出されない。この機能により、誤検出が削減される。

脆弱性データベースの拡張

脆弱性データベースは、言語毎に独立して用意される。データベースは XML 形式で記述されており、ユーザは独自にデータベースを作成することが可能である。以下では、データベースの記述について述べる。

データベースファイルには、ファイルの冒頭に下記の 2 点が記述されていることが推奨されている。

- XML 宣言
- DOCTYPE 宣言

これらの記述は以下のようにして行う。

```
<?xml version="1.0"?>
<!DOCTYPE RATS []>
```

上記の記述の他に、以下のようなルートエレメントの記述が行われていることが必須となっている。各脆弱性の記述は、ルートエレメント内にエレメントとして行う。以下のルートエレメントの記述では、C 言語が対象のデータベースとして宣言している。この為、RATS が C 言語以外の言語と判別した場合には、このデータベースが検査時に使われることはない。

```
<VulnDB lang="c">
```

データベース中の各脆弱性の記述については、以下の 2 点について記述されていることが必須である。

宣言の種類	使用するタグの例
各脆弱性の区切りの宣言	<Vulnerability> </Vulnerability>
名前の記述	<Name> </Name>

また、上記に加えて以下のような項目の記述も可能である。

宣言の種類	使用するタグの例
危険度	<Severity> </Severity>
脆弱性に付随する情報	<Info> </Info>
脆弱性の概要	<Description> </Description>
脆弱性の可能性のある引数の個所	<Arg> </Arg>、 <FormatArg> </FormatArg>、 <SrcBufArg> </SrcBufArg>
参照となる URL	<URL> </URL>
脆弱性の種類名	<RaceCheck> </RaceCheck>、 <RaceUse> </RaceUse>、 <InputProblem> </InputProblem>、 <FSProblem> </FSProblem>、 <BOProblem> </BOProblem>

以下に示すのは、標準の脆弱性データベースの一部である。

```
1  <Vulnerability>
2    <Name>strcpy</Name>
3    <BOProblem>
4      <SrcBufArg>2</SrcBufArg>
5      <Severity>High</Severity>
6    </BOProblem>
7  </Vulnerability>
```

図2.　脆弱性データベース例

データベースサンプルに記述されている内容は以下のようである。

行	解説
1 行目、7 行目	脆弱性の宣言毎に必ず記述するタグである。
2 行目	脆弱性の名前の記述を行う。記述する名前は検査対象となる関数名である。
3 行目、6 行目	脆弱性の種類の記述を行う。<BOProblem>はバッファーオーバーフロー問題を引き起こす可能性のある脆弱性に付けられるタグである。
4 行目	第何番目の引数について検査を行うかについて記述を行う。複数の引数について検査を行いたい場合は、別に<Vulnerability>タグで囲まれた脆弱性の定義を行う必要がある。
5 行目	危険度の記述を行う。危険度は High、Medium、Low の 3 種類で記述する。

検査の制御

RATS には、不要な検査を回避するために検査の制御機能が用意されている。この機能によって、明示的に特定の関数の検査の省略が可能である。この機能を使用するには、ユーザはソースコード中にコメントとコマンドを挿入する。

例：ソースコード中で以下の一文が書かれていたとする。

```
strcpy(buf, dst);
```

この行の検査を省略するには、以下のように記述すれば良い。尚、省略はコメントが挿入された行に限って有効である。

```
strcpy(buf, dst); // rats:ignore
```

検査省略の為にコメントを挿入する時には必ず、rats:または its4:という記述が入っていなければならない。用意されているコマンドについては、ignore 宣言のみである。これ以外の文字列が記入されると検査の省略が行われない。以下のような形式でコメン

トを書いた場合も検査の省略は可能である。

```
/* rats:ignore */
strcpy(buf, dst);
```

上記においては、省略はコメント文が挿入された次の行に対してのみ行われる。

結果の出力

検査結果は標準出力へ出力する。出力形式は、txt 形式・html 形式・xml 形式の 3 種類から選択することが可能であり、標準では text 形式で出力される。出力に含まれる情報は以下のような項目である。

- 使用した脆弱性データベースの情報
- 脆弱性の出現個所（ファイル名、行番号）
- 脆弱性の危険度（3 段階で表示）
- 検出された関数名
- 脆弱性の概要及び、その対策について
- テスト全体で要した時間
- 検査を行った行数の情報

標準では、出力される検査に要した時間等が付加情報として追加されているが、これらの情報の出力を行わないことが選択可能である。 html 形式での出力結果は以下のようになる。

図3.　HTML 形式での出力結果

開発環境との連携

開発環境と連携を行う機能は備えていない。

D） 開発体制

RATS の開発・メンテナンスはセキュアソフトウェア社によって行われている。開発にあたって、DARPA の CHATS プログラムから資金援助を受けている。開発メンバーである John Viega も DARPA より資金援助を受けている。John Viega は、RATS の開発を行う前には別のソースコード検査ツールである ITS4 の開発にも参加していた。この為、RATS の機能は ITS4 の機能と類似している。

現在、RATS の開発者は Flowfinder と相互でメンテナンスを行うことを目指している。脆弱性データベースはプロジェクト参加者以外の手でも作成されており、一部のデータベースはプロジェクトの中に取り込まれている。

E） 現在のバージョンと対応プラットフォーム

RATS の現在の最新バージョンは、2.1 であり、以下のサイトから入手可能である。

http://www.securesoftware.com/rats.php

RATS はインストールにあたって、expat がインストールされていることが必要である。expat は、通常は/usr/local/lib ディレクトリと/usr/local/include ディレクトリ以下にインストールが行われている。これ以外のディレクトリにインストールが行われている場合は、RATS の Configure スクリプトを実行する際にオプションを利用して、expat がインストールされているディレクトリの指定を行う。また、ソースコードは以下のサイトから入手可能である。

http://prdownloads.sourceforge.net/expat/expat-1.95.6-1.src.rpm?download

F） 利点と欠点

RATS では容易且つ高速に検査を行うことが可能である。出力結果の多くはデータベースとの機械的な照会の結果であるので、必ずしも正しいとは限らない。このため、出力結果の分析をする時には注意が必要である。場合によっては、数百にも及ぶ出力結果についての検証が必要である。RATS では、データベースの記述・拡張も容易に行えるよう考えられており、データベースの記述は XML 形式で記述するよう設計されている。これにより、ユーザは独自にデータベースを記述することが容易である。

出力は、xml 形式や html 形式を選択することが可能であり、ユーザが RATS の出力結果を分析しやすいよう設計されている。入出力やデータベースの記述のように、利用する

ユーザの便宜が図られたツールである。一方で、他の検査ツールと比較して、対応しているプログラミング言語の数が多いことから、C/C++以外の言語で書かれたソフトウェアについて、少しでも情報が欲しい場合にも利用できる。他の検査ツールでは、このような使い方はできないという点で、RATS の利点が挙げられる。

G） 対応言語

RATS は対応する検査可能なプログラミング言語が豊富であり、C、C++、Perl、Python、PHP の検査が可能となっている。この対応言語の豊富さは他の検査ツールでは見られない。

標準で付属するデータベースに含まれる脆弱性情報の個数の内訳を以下に示す。

言語	データベースの個数
C、C++	334
Perl	33
Python	62
PHP	55

表3. 脆弱性データベースの内訳

2.1.2. ITS4 に関する調査

A） 概要

ITS4（It's the software, stupid! Security Scanner）は、C/C++で書かれたソースコード検査が可能なソースコードの公開されたソフトウェアである。ITS4 の起動は Emacs 上でも可能であり、ユーザがソースコード閲覧及びコンパイル中でも起動しやすいよう考慮されている。脆弱性データベースには約 130 個の情報が格納されており、この情報に従ってソースコードの検査を行う。検査では、主にバッファーオーバーフロー問題とレースコンディション問題に対しての検出が可能である。ITS4 の開発者は、自らの監査業務の中で商用ソフトウェアの脆弱性の検出に利用を行っており、この検出結果についての考察を論文に記述している。

B） 検出可能な脆弱性

ITS4 において検査の対象となるのは、主に以下の項目についてである。

(1)バッファーオーバーフローを引き起こす可能性の高い関数

例：fgets, gets, bcopy, fgetc, fscanf, getc, getchar, getopt, getopt_long, getpass, memcpy, read, scanf, snprintf, sprintf, sscanf, strcat, strccpy, strcpy

(2)レースコンディションを引き起こす可能性の高い関数

例：access, acct, basename, catopen, chdir, chgrp, chmod, chown, chroot

(3)乱数生成に関する関数

例：drand48, erand48, jrand48, lrand48, mrand48, nrand48, rand, random

(4)ユーザが悪意のあるコードを埋め込む可能性のある関数

例：fread, recvmsg, recvfrom, recv

(5)フォーマット・ストリング・バグの可能性の高い関数

例：fprintf, fwprintf, printf, vfwprintf, vwprintf, wprintf

C） 機能詳細

ITS4 の機能を次の 6 つの観点で解説する。

・データ入力
・ソースコードの検査
・脆弱性データベースの拡張
・検査の制御
・結果の出力
・開発環境との連携

<u>データ入力</u>

ユーザが ITS4 に引き渡すことができるデータは、検査対象となるソースコードと脆

弱性データベースの二つである。

①検査対象となるソースコードの読み込み

ユーザが検査対象として指定したファイル名を元に、ソースコードの読み込みを行う。ディレクトリ名の指定による検査は行えない。ディレクトリ以下のファイルの一括検査を行う場合には、シェルの正規表現を用いてファイル名の指定を行う。ITS4 には、正規表現を用いて検査ファイルを指定した際に、検査を行わないファイル名を指定する機能がある。RATS のように自動的にファイル拡張子を判断する機能は備えていない。

②脆弱性データベースの指定

ユーザが独自に作成を行った脆弱性データベースを指定することが可能である。脆弱性データベースを指定した場合も標準で用意されている脆弱性データベースが読み込まれる。

ソースコード検査

検査は、脆弱性データベースとソースコード中で使用されている関数のパターンマッチングによって行われる。検査の精度の向上・効率化のために次のような機能が用意されている。

①データベースを用いたパターンマッチング

ITS4 は、起動時に脆弱性データベースの読み込みを行う。脆弱性データベースの情報は、読み込まれた順にメモリ領域にスタックされる。スタック情報に格納された情報と、ソースコード中の記述の比較によって検査を行う。比較はソースコードの先頭から行われる。比較を行う情報は、脆弱性データベースに記述された関数名とソースコード中で宣言・使用された関数名である。脆弱性データベースの情報とソースコードの記述の比較を効率的に行うために、脆弱性データベースの読み込み時に関数名情報のハッシュを行っている。この結果、関数名の比較を効率的に行うことが可能である。また、ITS4 はこのハッシュを用いた比較の処理を効率的に行う実装をしており、RATS より効率的な検査を実現している。以下にパターンマッチングの概要をまとめた図を示す。

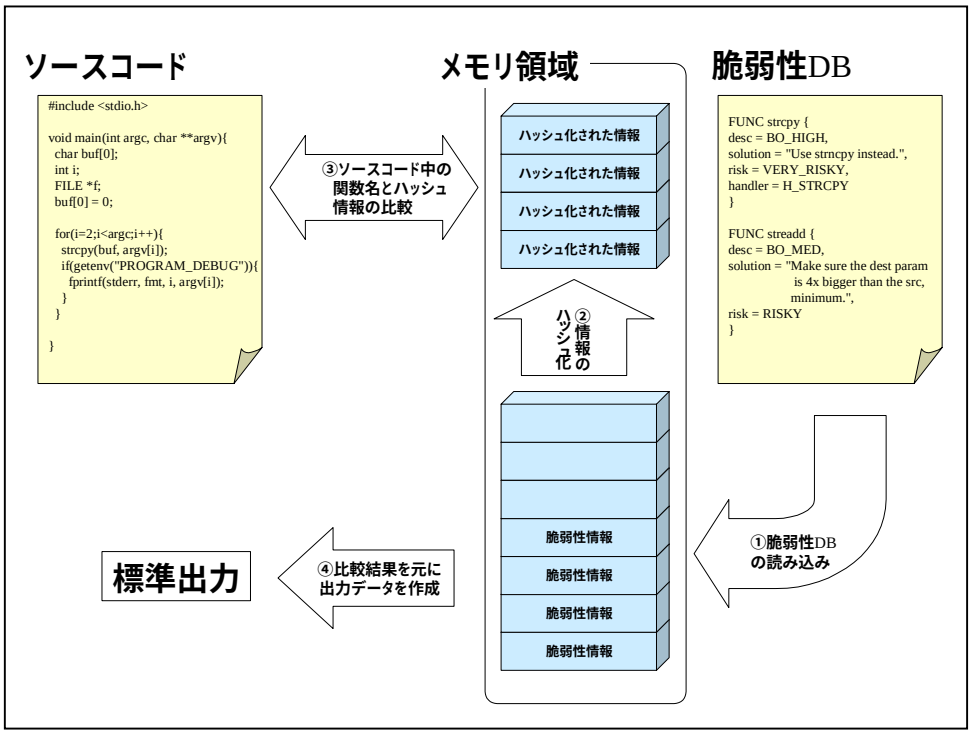

図4.　ITS4 の動作

②構文解析による変数名と関数名の識別

脆弱性データベースに登録された関数名と同じ名前の変数を宣言した場合、ITS4 は検出を行わない。これは、内部で簡単な構文解析を行っているためである。この構文解析によって、誤検出が少なくなる。構文解析の精度は RATS の方がより詳細に行うことが可能である。構文解析は以下のようにして行っている。

ⅰ：脆弱性データベースに関数名に伴って、引数の個数の指定を行う。

ⅱ：ITS4 は読み込んだソースコード中に、名前が合致する個所を見つけ出す。

ⅲ：合致した個所に“（”と“）”が記述されているかを検査する。

解析の判断を以下のようなコードに対して、確認をする。

```
1　int printf;
2　int printf();
3　int printf(“test1”);
```

上記では、printf という文字列を 3 種類記述している。この記述が検出されるかの確認を行う。以下は標準の脆弱性データベースにおける printf 関数の記述である。

```
FUNC printf {
desc = FORMAT_DESC,
solution = FORMAT_SOL,
risk = LOW_RISK,
handler = H_PRINTF
}
```

上記のデータベースを用いて検査を行った結果は以下である。

コード	検出の可否
1 行目	×
2 行目	○
3 行目	○

1 行目における宣言では、“ () ”が使われていないために、関数として判断されない。2 行目おける宣言では、“ () ” が使われているため関数として検出された。3 行目では、“ () ”が使われているため、引数として渡される文字列によらず、printf 関数が使用されていると検出される。

③コメント文・マクロ文の検査対象からの除外

コメント文に記述された関数名については、パターンマッチングは行われない。コメント文に対しては、後述の検査の省略にあるように、ITS4 特有のコマンドを挿入可能である。従って、コメント文はソースコードの検査対象としては扱われないが、検査の効率化の対象としては扱われる。

マクロ文に関しての検査も行われない。マクロ文で脆弱性の可能性の高い関数名が変換されている場合には注意が必要である。マクロ文で脆弱性の可能性の高い関数名が変換されている場合には、変換後の記述形式を脆弱性データベースに登録することで、検査可能である。

④安全な引数の判別

ITS4 では、幾つかの関数（strcpy 等）の検査において、安全な引数の引渡しが行われる場合には、脆弱性の可能性のある関数を使用しているが安全な使用をしているとして脆弱性の検出を行わない機能がある。例として、以下のような記述に対して検査を行う。

```
1   strcpy(buf, “test¥n”);
2   strcpy(buf, argv[1]);
```

2 行目において、ユーザが buf のサイズより大きい値を入力することができるため脆弱性の可能性が検出される。一方で 1 行目においては、内部で定義された文字列

であることから脆弱性の可能性は検出されない。この機能により、誤検出が削減される。

⑤レースコンディションへの対応

ITS4 はレースコンディションの検査への対応が充実している。レースコンディションの検査を行うために、外部からのファイル入出力のある関数とその関数の引数を検査対象とする。検査を確実に行うために、ファイルの入出力に関わる情報をメモリの中に格納し、検査結果を出力する前に再度レースコンディションに関わる検査を行う。

脆弱性データベースの拡張

ユーザは独自に脆弱性データベースを作成することが可能である。以下では、脆弱性データベースの記述について述べる。

標準の脆弱性データベースには、開発者の経験に基づいた脆弱性情報の他に Bugtraq アーカイブから選定された脆弱性情報が約 130 個登録されている。このデータベースで大きな割合を占めているものの一つには、バッファーオーバーフローに関係した関数の情報である。また、レースコンディション問題に関係した関数の情報も多く含まれている。標準で用意されたデータベースは以下のような項目から構成されている。

① 各脆弱性で共通に参照される情報
② 各脆弱性に関する記述

共通に参照される情報として、以下のような情報が格納される。

① 問題の概略
② 解決方法
③ 危険度の分類

危険度は、NO_RISK、LOW_RISK、MODERATE_RISK、RISKY、VERY_RISKY、MOST_RISKY の６段階に分かれている。この危険度に基づいて出力結果はソートされる。ユーザは、各危険度の分類を変更することが可能となっているが、推奨はされていない。

④ 外部からの入力が行われる関数での入力についての情報

標準で用意されたデータベースでは、上記の情報を各脆弱性に関する記述の中で参照している。この参照情報とは独立に、各脆弱性で、以下のような記述が行わなければならない。

① 脆弱性のある関数名

② 脆弱性の概要
③ 対策方法に関する記述

上記を踏まえた各脆弱性のデータベースの記述スタイルは以下のようになっている。

```
FUNC “関数名” {
“脆弱性に関する記述（概要･対策･危険度･引数に関する記述）”
}
```

システムに付属している脆弱性データベースの一部を以下に掲載する。

```
1    FUNC strcpy {
2            desc = BO_HIGH,
3            solution = "Use strncpy instead.",
4            risk = VERY_RISKY,
5            handler = H_STRCPY
6    }
```

データベースサンプルに記述されている内容は以下のようになっている。

行	解説
１行目	脆弱性を持つ可能性のある関数の名前の記述を行っている。
２行目	脆弱性の概要についての記述を行っている。
３行目	脆弱性に対する対策方法の記述を行っている。
４行目	危険度を表記している。
５行目	関数の種類を表記している。この類別により、関数に引き渡される引数のチェックの内容が決定される。

検査の制御

ユーザがソースコード中にコメントとコマンドを挿入することで､明示的に特定の関数の検査を省略が可能である。

例：ソースコード中で以下の一文が書かれていたとする。

```
strcpy(buf, dst);
```

この行の検査を省略するには、以下のように記述すれば良い。尚、省略はコメントが挿入された行に限って有効である。

```
strcpy(buf, dst); // its4:ignore
```

検査省略の為にコメントを挿入する場合、必ず its4:という記述が入っていなければならない。同様に its4 の記述の後には必ず、コマンドが入っていなければならない。コ

マンドが入っていない場合、ITS4 は異常終了する。

用意されているコマンドは、ignore 宣言のみである。これ以外の文字列が記入されると検査の省略が行われず、通常通りの検査が行われる。また、以下のような形式でコメントを書いた場合も検査の省略は可能となる。

```
/* its4:ignore */
strcpy(buf, dst);
```

上記においては、省略はコメント文が挿入された次の行に対してのみ行われる。

結果の出力

検査結果を標準出力または引数で指定したファイルへ出力する。出力形式は、txt 形式と MS-Visual Studio 形式から選択可能である。出力に含まれる情報は以下のような項目についてである。

- 脆弱性の出現個所（ファイル名、行番号）
- 脆弱性の危険度（6 段階で表示）
- 検出された関数名
- 脆弱性の概要
- 脆弱性に対する対策方法

出力は標準出力に対して行われる。オプションの指定により、出力先を標準出力からファイルに変更することも可能である。出力される脆弱性のソート方法もオプションで指定することが可能である。このソートの機能から、危険度の高い脆弱性を優先的に出力することが可能である。また、Microsoft Visual Studio 形式に対応した出力も可能である。

開発環境との連携

ITS4 は、Emacs、及び Microsoft Visual Studio を使用している開発者が容易に使用できるよう考慮されている。Emacs 上で ITS4 を利用するには、以下のような記述を.emacs に追加する。

```
;   A quick elisp hack piggybacking off compile mode so you can
;   hit <enter> over an error to jump to the line in the source.
;   Use "M-x its4-scan" to run.   If its4 isn't in your path, change
;   the value of its4-location to point to an absolute path.
(setq its4-location "its4")
(require 'compile)
(defun its4-scan (args) (interactive "sRun its4 with args: ")
   (compile (concat its4-location " " args))
)
```

D） 開発体制

ITS4 は、米国 cigital 社によって開発・配布が行われている。ITS4 の初期の開発メンバーである John Viega は、DARPA より資金援助を受けており、その成果物の一つが ITS4 である。現在、John Viega はソースコード検査ツールである RATS の開発プロジェクトに取り組んでいる。ITS4 の開発プロジェクトは現在の所、他の言語への拡張は行わない方針になっており、細かいバグの修正に注力している。また、ITS4 はソースコードが公開されているが、ライセンス形態がオープンソース・ソフトウェアとして公開されている他の検査ツールと異なり、非商用利用に限ってフリーで利用することが可能である。

E） 現在のバージョンと対応プラットフォーム

ITS4 の最新バージョンは 1.1.1 であり、以下のサイトから入手が可能である。

http://www.cigital.com/its4/

インストール時に他のライブラリ、ソフトウェアは必要としない。

F） 利点と欠点

ITS4 は、RAT と同様に検査結果として数百行にも及ぶ出力を行うことがあり、検査結果についての吟味が必要となる。基本的な機能特性は、RATS と酷似している。

RATS と比較して勝っている個所は、IDE 環境への対応が考慮されている点である。一方で、RATS に劣っている点は、対応しているプログラミング言語の数である。検出できる脆弱性の範囲については、RATS と決定的な差はない。

ITS4 で特筆すべき点は MS 製品関連の Lib 関数における脆弱性の情報を、データベースに登録していることである。脆弱性データベースのカスタマイズは、比較的容易に行うことができ、脆弱性の基本的な項目についての記述だけを行っても十分に検査が行える。一方で、脆弱性の詳細な記述を行うには、ITS4 のソースコードを読む必要があり、決して容易ではない。

G） 対応言語

C、C++に対応している。

2.1.3. Flawfinder に関する調査

A） 概要

Flaswfinder は Dr. David A. Wheeler によって開発されているオープンソース・ソフトウェアである。GPL の規約の元でコピー・再配布等が許可されている。C/C++のソースコードの検査を行うことができ、主な検査項目としては、バッファーオーバーフローやレースコンディションの脆弱性が挙げられる。検査可能な脆弱性項目は約 130 個用意されている。

Flawfinder の記述言語は Python であり、脆弱性データベースは Python スクリプト中に取り込まれている。Flawfinder ではユーザが外部に作成したデータベースを実行時に取り込むことはできない。この為、新たな記述を行いたい場合は、ソースコード中のデータベースの個所を直接編集する必要がある。また、Flawfinder では他のツールでは検査を考慮されていなかった gettext 関数郡についても解析が可能となっている。プログラムの外部からデータを取得する関数のみの検査を行うことも可能である。これによって、脆弱性の発見・攻撃が行われやすい個所を重点的に検査することが可能となる。

B） 検出可能な脆弱性

Flawfinder において検査の対象となるのは、主に以下の項目についてである。

(1)バッファーオーバーフローを引き起こす可能性の高い関数

例：strcpy(), strcat(), gets(), sprintf(), scanf()

(2)フォーマットストリングの脆弱性を生みやすい関数

例：printf(), snprintf(), syslog()

(3)レースコンディションを引き起こす可能性の高い関数

例：access(), chown(), chgrp(), chmod(), tmpfile(), tmpnam(), tempnam(), mktemp()

(4)潜在的なシェルメタキャラクタの使用の可能性

例：exec(), system(), popen()

(5)乱数生成に関する関数

例：random()

C） 機能詳細

Flawdinder の機能を次の６つの観点で解説する。

・データ入力
・ソースコードの検査
・脆弱性データベースの拡張
・検査の制御
・結果の出力
・開発環境との連携

データ入力

ユーザが Flawfinder に引き渡すことができるデータは、検査対象となるソースコードと過去の検査結果の２つである。

①検査対象となるソースコードの読み込み

C/C++のソースコードのファイル名をコマンド引数で受け取る。ディレクトリ名をコマンド引数に渡して、ディレクトリ以下のファイルを検査することが可能である。ディレクトリ以下のファイルの検査を行う場合、Flawfinder はファイルの拡張子を判別して、C/C++のソースコードだけを検査する。標準では、シンボリックリンクは検査の対象外となっている。シンボリックリンクの検査を許容することは、シンボリックリンク攻撃の可能性があり、これは Flawfinder の安全な動作を妨げることになるからである。

②出力結果の読み込み

Flawfinder には、検査結果の出力を保存する機能があり、保存された出力を再度読み込み最新の検査結果と比較する機能もある。この機能により、ソースコードの修正前と修正後で、どの程度脆弱性の可能性が減ったかが把握可能である。

ソースコード検査

①データベースを用いたパターンマッチング

Flawfinder は、脆弱性データベース記述とソースコード中に使用されている関数との比較を行う。比較はソースコードの先頭から行われる。比較の処理には特別な処理は用意されていない。

以下に Flawfinder のパターンマッチングの概要をまとめた図を示す。

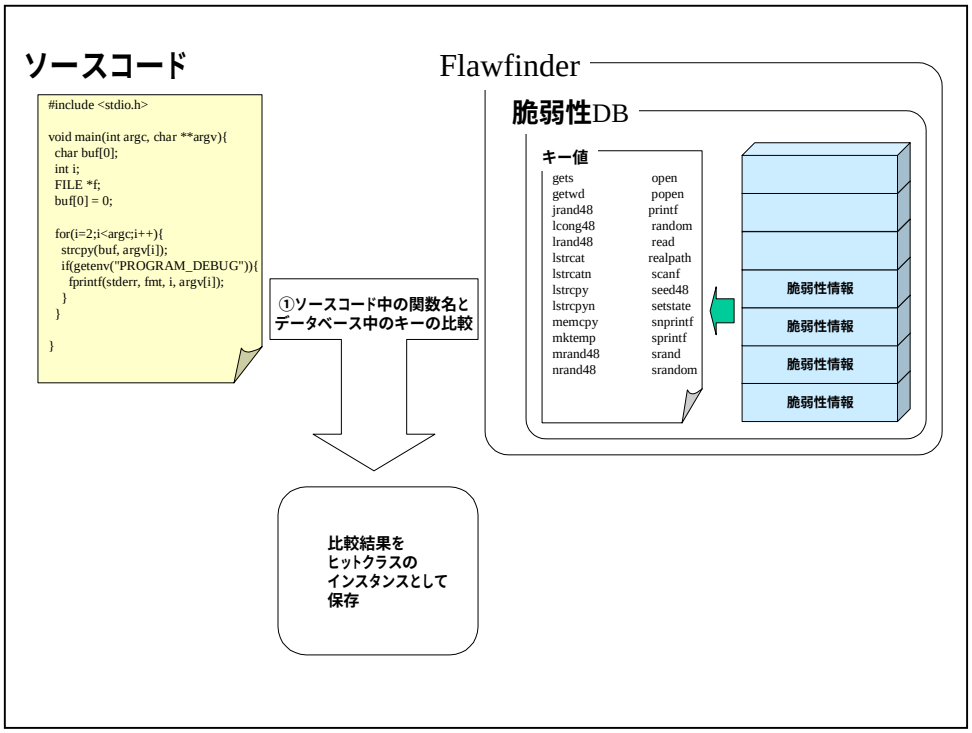

図5. Flawfinder の動作

②コメント文の検査対象からの除外

コメント文に記述された関数名については、パターンマッチングは行われない。コメント文に対しては、後述の検査の省略にあるように、Flawfinder 特有のコマンドを挿入可能である。

③マクロ文のマッチング

マクロ文を構造的に検査する機能はない。しかし、マクロ文に脆弱性データベースとマッチする記述があった場合には、検出される。これは、Flawfinder がマクロ文の記述の検査を除外する機能を持たないためである。

④サイズの指定されたバッファの宣言の検出

Flawfinder は ITS4 と異なり、サイズの指定されたバッファの宣言の検出の検出を行う。この機能は RATS では実装されている。

⑤外部からデータ入力のある関数の検出

外部から入力のある関数（read、scanf 等）が存在するかどうかのみを検査する機能がある。プログラムにおいて、ユーザからの入力が行われる個所が最も危険性が

高い。従って、入力個所の脆弱性を把握・軽減することで、ソースコードに潜んだリスクを軽減することが可能である。

脆弱性データベースの拡張

脆弱性データベースは、Flawfinder の実体である Python スクリプトに記述されている。この為、ユーザが外部のファイルにデータベースを作成し、それを利用した検査を行ことはできない。一方でユーザは、Flawfinder 内のデータベースを直接編集することは可能であるが、正しい記述が行えないと正常な動作を損なう可能性がある。

Flawfinder に付属する脆弱性データベースには約 130 個の脆弱性を関する情報が含まれている。脆弱性データベースに登録された関数名を閲覧する機能がある。

検査の制御

ユーザがソースコード中にコメントとコマンドを挿入することで､明示的に特定の関数の検査を省略できる。

例：ソースコード中で以下の一文が書かれていたとする。

```
strcpy(buf, dst);
```

この行の検査を省略するには、以下のように記述すれば良い。尚、省略はコメントが挿入された行に限って有効である。

```
strcpy(buf, dst); // flawfinder:ignore
```

ユーザがソースコード中にコメントとコマンドを挿入することで､明示的に特定の関数の検査を省略できる。

検査省略の為にコメントを挿入する時には必ず、flawfinder:、rats:、its4:という記述が入っていなければならない。用意されているコマンドについては、ignore 宣言のみである。この為、現在のバージョン（Ver.1.21）では、ignore の記述を行わなくとも、検査を省略することが可能である。

以下のような形式でコメントを書いた場合も検査の省略は可能となる。

```
/* flawfinder:ignore */
strcpy(buf, dst);
```

上記においては、省略はコメント文が挿入された次の行に対してのみ行われる。オプションの指定でコマンドの省略を無視することが可能である。

結果の出力

Flawfinder は、検査結果を標準出力または引数で指定したファイルへ出力する。出力形式は、txt 形式と html 形式から選択できる。出力に含まれる情報は以下のような項目についてである。

- 脆弱性の出現個所（ファイル名、行番号）
- 脆弱性の危険度（6 段階で表示）
- 検出された関数名
- 脆弱性の概要と対策について
- 発見された脆弱性の個数
- 検査を行ったソースコードの行数と検査に要した時間

出力は標準出力に対して行われる。オプションの指定により、出力先を標準出力からファイルに変更することが可能である。標準では、出力は危険度が高いものから行われる。この危険度の判断は関数だけに依存しておらず、関数の引数にも依存している。この為、ユーザはより精度が高い状態で、危険度の高い関数の使用を優先的に知ることが可能である。

出力の形式は txt 形式と html 形式から選択できる。html 形式の出力を Web ブラウザで表示すると以下のようになる。

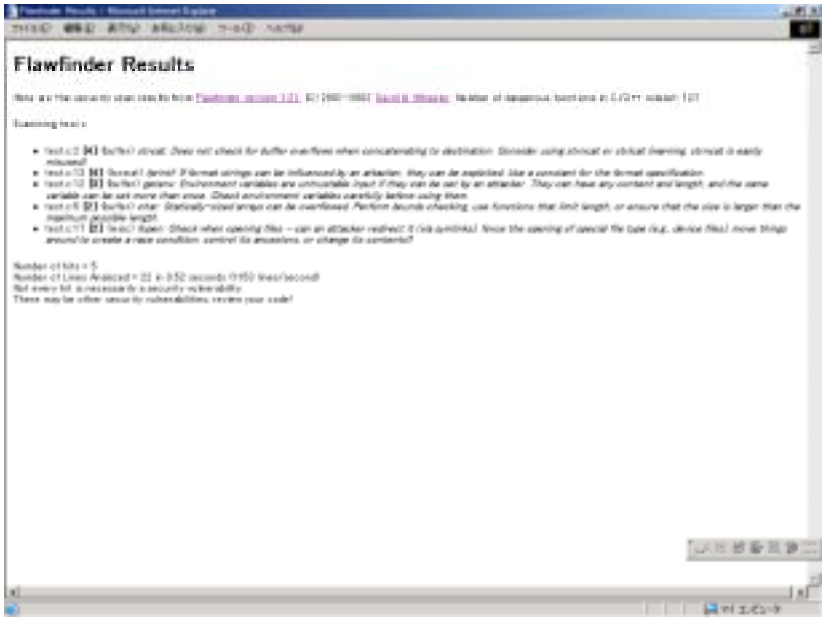

図6.　HTML 形式での出力結果

開発環境との連携

開発環境と連携を行う機能は備えていない。

D） 開発体制

Flawfinder は、Dr. David A. Wheeler によって開発が行われている。開発者である Dr. David A. Wheeler は、「Secure Programming for Linux and Unix HOWTO」の著者としても知られている。開発にあたっては、有志の手によってパッチが提供されており、随時反映が行われている。Flawfinder は、同様なソースコード検査ツールである RATS との相互メンテナンスを行うことを目指している。双方のプロジェクトは検査の観点で酷似しており、ツールの利用者にとっても二つのそれぞれの視点でソースコードにコメントが付くのは有益である。このような議論の結果、RATS と Flawfinder は相互メンテナンスを目指すことになった。また、同様に検査の観点とアプローチが類似している ITS４については、オープンソース・ソフトウェアではないという理由から、現在の所、共同で作業を行う予定はない。

E） 現在のバージョンと対応プラットフォーム

Flawfinder の最新バージョンは 1.21 であり、以下のサイトから入手できる。

http://www.dwheeler.com/Flawfinder/

インストール時に必要となるのは Ver.1.5 以降の Python のみである。この要件さえ満たしていれば様々な環境上での利用が可能である。

F） 利点と欠点

Flawfinder でも、出力結果は数百行にも及ぶことがあり、この点の議論は RATS や ITS4 と同様である。Flawfinder が他のパターンマッチング型の検査ツールと比較して勝っているのは、多国語言語関数である gettext 関数郡に対応していることである。詳細な個所で開発者の経験に基づく実装が行われている。例えば、出力結果に改行コードを挟まない、検査対象となるファイルにはシンボリックリンクを含まない等である。この結果、検査精度は高い。

脆弱性データベースはソースコードに組み込まれており、ユーザが容易に追加・変更することができない。この点で柔軟性が損なわれている。また、Python スクリプトで記述が行われている為に、C 言語で記述された他のツールと比較して速度は遅い。他のツールと比較すると遅いという結果になるが、実用面では必ずしも遅いとはいえないレベルの実装が行われている。

G） その他（対応言語）

C、C++に対応している。

2.2. 構文解析技術

構文解析技術とは、プログラマが lint コマンドを使ってプログラムミスやバグを把握する作業を模倣した技術である。構文解析型では、ソースコード中のデータ構造の遷移等を解析することで、論理的な矛盾点を見つけ出す。幾つかの脆弱性は単純なプログラムミスから発生していることから、ソースコードが論理的に厳密な記述されることで、セキュリティ上の信頼度は高められるという視点に立っている。構文解析技術を採用した検査ツールでは、通常の構文解析技法に新たにセキュリティ上の検査を行うための機能が追加されている。本章では、構文解析技術のツールとして、Splint、Cqual の２件について報告をする。

2.2.1. Splint に関する調査

A） 概要

Splint は米国の Virginia 大学で開発が行われている静的なソースコードチェックを行うことができるオープンソース･ソフトウェアである。ライセンスは、GPL である。Splint は、C 言語の構文解析ツールである LCLint がベースになっており、セキュリティの検査を行うだけでなく、文法の検査も行うことが可能である。コメント文に Splint 特有のコマンドを記述することで、通常の構文解析の機能を拡張している。コマンドを記述することによって、使われていない変数宣言や、型宣言の衝突、未宣言の変数、戻り値の型の不一致、無限ループ等の検査を精密に行うことができる。このコマンドの記述が Splint の検査機能の特徴である。コマンドの記述は柔軟になっている反面、詳細な検査を行うには対象となるアプリケーションに対する知識と詳細なコマンドの記述が必要となる。コマンドの記述を行わなくとも、標準の状態で検査を行うことが可能である。

Splint は Win32 上や OS/2 上で動作もする。また、Linux 向けに RPM 形式も配布されており、多彩なプラットフォーム上でも動作する。

B） 検出可能な脆弱性

Splint において検査の対象となるのは、主に以下の項目についてである。

(1)不正な null pointer への参照
(2)未定義の記憶領域の使用、適切でない定義の記憶領域へのリターン
(3)型の不一致
(4)メモリ・リークや Dangling reference の使用を含む、メモリ管理エラー
(5)関数のインタフェースと矛盾する変数の使用
(6)無限ループや網羅されていない case 構文の使用といった問題の可能性が高い実装
(7)バッファーオーバーフロー脆弱性
(8)危険なマクロの記述
(9)変数・関数の命名規則違反

その他に Splint では、上記の検査機能だけに留まらず、lint の機能も有しており、文

法上の誤りといった、基本的な事項についての検出が可能である。

C） 機能詳細

Splint は構文解析技術をベースにしたソースコード解析ツールである。既存の構文解析ツールの機能に、ソースコードのセキュリティ検査機能が加えられている。セキュリティ検査機能は、コメント文に Splint 特有のコマンドを記述することで実現している。コメント文には、以下のような形式でコマンドの記入を行い、従来のソースコードの記述を崩すことなく検査を行うことが可能である。

```
/*@（コマンドをここに記述する）@*/
```

上記のような記述で、様々な情報・コマンドが記述できることによって、設計段階では考慮されてないプログラムの状態遷移について検査可能である。また、マクロのチェックやライブラリ参照の妥当性についても検査可能である。

標準ライブラリ関数の記述についての対応も行われており、対応が行われているのは、ISO、ANSI、POSIX の標準ライブラリである。以下では、Splint の機能を次の 3 つの観点で解説する。

・データ入力
・ソースコードの検査
・検査ルールの拡張
・抽象構文木
・結果の出力
・開発環境との連携

データ入力

ユーザが Splint に引き渡すことができるデータは主に、検査対象となるソースコードとヘッダファイルの２つである。

①検査対象となるソースコードの読み込み

Splint は起動時にソースコードの読み込みを行う。ディレクトリ名の指定によるソースコードの読み込みを行う機能は有していない。ディレクトリ以下のファイルの一括検査を行う場合には、シェルの正規表現を用いてファイル名の指定を行う。ファイル名の拡張子で、読み込むファイルを振り分ける機能は有しておらず、正規表現を用いた一括検査を行う場合には注意が必要である。

②ヘッダファイルの読み込み

Splint は起動時にヘッダファイルの読み込みを行う。ヘッダファイルは、ソース

コード中に呼び出される関数の妥当性を検査するために必要とされる。Splint はソースコード中でヘッダファイルを読み込む記述がされていた場合、予め指定されている検索パスに従ってヘッダファイルを検索する。検索の結果、ヘッダファイルが見つからない場合には Splint は処理を中断する。読み込まれるヘッダファイルが標準的なディレクトリ以下に保存されていない場合、保存されているディレクトリの情報を明示的に Splint に渡さなければならない。

ソースコード検査

Splint の基本的な検出方法は、ソースコードのコメントに埋め込んだコマンドを利用して、プログラムの仕様を記述し、その仕様の制約にマッチしないコードが存在するかどうかをチェックする。そのためソースコード中には、そのコードにおいて要求される制約を記述しなければならない。Splint はさまざまなチェックが可能であるが、ここでは以下の 3 つの検出機能について説明する。

①不正な null pointer の参照の検出

null pointer の参照によって、プログラムのエラーはしばしば発生する。Splint はこれらのエラーを検出することが可能である。以下のような例で説明する。

```
1  char firstChar1 (/* @null@*/ char *s)
2  {
3        return(*s);
4  }
5  char firstChar2 (/*@null@*/char *s)
6  {
7        if (s == NULL) return'¥0';
8        return(*s);
9  }
```

/* @null@ */は引数の値が NULL である可能性を示す Splint の定義である。この定義が行われている場合、Splint は定義された変数が null かどうかのチェックを行われずに利用されているプログラムの問題を検出する。上記の例では 3 行目は変数 s が NULL かどうかをチェックせずに参照しているため検出される。逆に、9 行目は直前の 8 行目で変数 s が NULL かどうかをチェックしているために、Splint では安全だと判断し、検出しない。

②バッファーオーバーフローの脆弱性の検出

Splint ではバッファーオーバーフローが発生しそうなエラーについて検出可能である。Splint のバッファーオーバーフローの検出モデルでは、maxSet と maxRead という 2 つのプロパティが利用される。バッファ b が与えられた場合に、maxSet(b)はバッファ b を安全な利用が可能なメモリ上限を示す。例えば char

buffer[MAXSIZE]では maxSet(buffer) = MAXSIZE – 1 となる。同様に maxRead は安全な利用が可能なバッファのインデックス上限を示す。この定義では NULL で終了しない文字列に対して上限を超えて読み出しを行うようなエラーに対して対処可能である。上記の検出モデルに従って、脆弱性のある標準化関数である strcpy を以下のように定義する。

```
1  void /* @alt char * @*/ strcpy
2  (/* @unique@ */ /*@out@*/ /*@returned@*/ char *s1, char s2)
3  /*@modifies *sl@*/
4  /*@requires maxSet(sl) >= maxRead(s2) @*/
5  /*@ensure maxRead(s1) == maxRead(s2)@*/
```

上記の 4 行目の require という記述は s1 として渡されるバッファが s2 として渡されるバッファよりも大きいか等しくなければならないことを示している。5 行目の ensure はこの関数呼び出しの後に s1 の maxRead が s2 の maxRead と同じになることを定義している。上記では、s2 のサイズは不定となるので、strncpy 関数に対して以下のように定義する。

```
1  void /* @alt char * @*/ strncpy
2  (/* @unique@ */ /*@out@*/ /*@returned@*/ char *s1, char s2, size_t n)
3  /*@modifies *sl@*/
4  /*@requires maxSet(sl) >= (n – 1 ); @*/
5  /*@ensure maxRead(s1) == maxRead(s1) /¥ maxRead (s1) <= n; @*/
```

上記の定義を用いて、以下のようなソースコードに対して検査を行う。

```
1   void updateEnv(char * str)
2   {
3         char * tmp;
4         tmp = getenv(“MYENV”);
5         if(tmp != NULL)
6               strcpy(str, tmp);
7   }
8
9   void updateEnvSafe(char *str, size_t strsize)
10                /* @requires maxSet(str) >= strSize – 1@ */
11  {
12        char * tmp;
13        tmp = getenv(“MYENV”);
14        if (tmp != NULL)
15        {
16              strncpy(str, tmp, strSize –1);
17              str[strSize –1] = ‘¥0”;
18        }
19  }
```

以下は、上記のソースコードに test.c というファイル名を付け、検査を行った結果である。

```
bounds.c:6: Possible out-of-bounds store:
   strcpy(str, tmp)
   Unable to resolve constraint:
   requires maxSet(str @ test.c:6) >= maxRead(getenv (“MYENV”) @ bounds.c:4)
    needed to satisfy precondition:
   requires maxSet(str @ bounds.c:6) >= maxRead(tmp @ bounds.c:6)
    derived from strcpy precondition: requires
   maxSet(<parameter 1>) >= maxRead(<parameter 2>)
```

上記のように、test.c の 6 行目の strcpy において、第 2 引数のバッファサイズが第 1 引数のバッファサイズを越える可能性があることが警告されている。

Splint では、前述した strcpy の制約条件を持っており、この制約条件が満たされない場合を検出する仕組みをとっている。それに大して 16 行目は検出されていない。これは、strncpy の制約が満たされているためである。例では、str のサイズは strSize-1 以下であることが定義されている。そのため、strncpy 関数での str バッファへの書き込み制約を満たしており、エラーとならない。

③危険なマクロの記述

Splint では、マクロの記述についての検査機能も有している。マクロの記述の検査機能を実装しているのは、今回調査を行った 5 つのツールの中では Splint だけである。

Splint はマクロ展開を行い、展開後のコードの検査を行う。マクロ展開後のコードに対して、マクロの記述で取りうるパラメータの代入の検査を行う。この検査によって、設計段階では考慮されていない不正なデータの代入が検出可能である。

マクロ記述に対して Splint のコマンドの記述を行っている場合、Splint はコマンド毎にマクロの検査範囲の限定を行い、コマンドの記述によってはマクロ展開を行わずに検査を行う。マクロは展開が行われなかった場合には、あたかも変数の一種のように取り扱われる。

マクロの検査のコントロール機能も備えており、例えば/*@notfunction@*/コマンドは、関数として振る舞うマクロの検査を行って欲しくない場合に付加する。このコマンドにより、Splint は記述されたマクロの構文的な意味の検査は行わず、文字列として扱い検査を行う。マクロを用いた定数表現の検査を行うことが可能である。以下のような宣言を例として説明する。

```
#define MinValue 0
```

Splint は上記の記述があった場合には、数値しては捉えない。そこで、以下のような記述を行う。

```
/*@constant int MinValue;@*/
```

```
#define MinValue 0
```

この結果、Splint は展開される MinValue は int 型として解釈し、int 型として扱うことのできない個所で MinValue が使われた場合には、警告を行う。

検出ルールの拡張

Splint ではユーザは自由に検出ルールを拡張することが可能である。検出ルールを拡張するためには、.mts 拡張子のファイルに独自の検出ルールを記述する必要がある。以下に検出ルールの例を示す。

```
1   attribute taintedness
2       context reference char *
3       oneof untainted, tainted
4       annotations
5           tainted reference ==> tainted
6           untainted reference ==> untainted
7       transfers
8           tainted as untainted ==> error "Possibly tainted strage used where ¥
             untaineted required"
9       merge
10           tainted + untainted ==> tainted
11       defaults
12           reference ==> tainted
13           literal ==> untainted
14           null ==> untainted
15   end
```

上記の例では、taintedness という制約を定義している。taintedness は char * のバッファが信頼できない情報源からのものなのかどうかを追跡するためのものである。これは、フォーマット・ストリング・バグ攻撃の可能性等の検出に利用可能である。

はじめの 3 つの行では char * 型のオブジェクトと結びついて taintedness の属性情報を定義している。3 行目において、この制約がとりうる 2 つの状態（tainted と untainted）を定義している。2 行目の context という定義では、char * 型の変数が taintedness と関係付けられることを定義している。3 行目の one of は taintedness において定義可能な 2 つの限定子を定義する。untainted は信頼置けない入力からのものではないものを指し、tainted は悪意を持った入力を指す。4 行目の annotations は 2 つの新しい注釈を定義する。reference==>という表現は tainted という文字列が taintedness の tainted 状態であることを示すことを定義するものである。7 行目の transfer は状態の遷移とその場合のエラーの内容について定義するものである。ここでは tainted から untainted への遷移はエラーとなることが定義され、そのときのエラーメッセージが定義されている。その他の遷移は暗黙的に許可されることになる。9 行目の merge は遷移が長いパスとなった場合の、遷移状

態のマージ方法について定義する。ここでは、tainted+untainted は tainted であることを示している。11 行目の default は明示的なアトリビュート宣言がない場合のデフォルトの値を定義している。ここでは、reference、literal、null をそれぞれ tainted、untainted、tainted に定義している。

これらの定義は、Splint 実行時に標準のルールに追加して実行することができる。

抽象構文木

Splint はソースコードの検査にあたっては、内部で抽象構文木を作成する。この構文木に基づいて、プログラムの正当性の検査を行う。検査にあたって、最初に各関数と関数における変数の入出力についての解析を行う。各関数における解析で、検査の対象となるのはグローバル変数と関数の引数となる変数である。各関数についての解析結果を元に、プログラムの実行の流れの中での関数同士の連携について検査する。これらの一連の検査に加えて、Splint では前述のようにコメント文へのコマンドの記述とそれらの解析によって、セキュリティ面での検査を強化している。以下に Splint の検査の概要をまとめた図を示す。

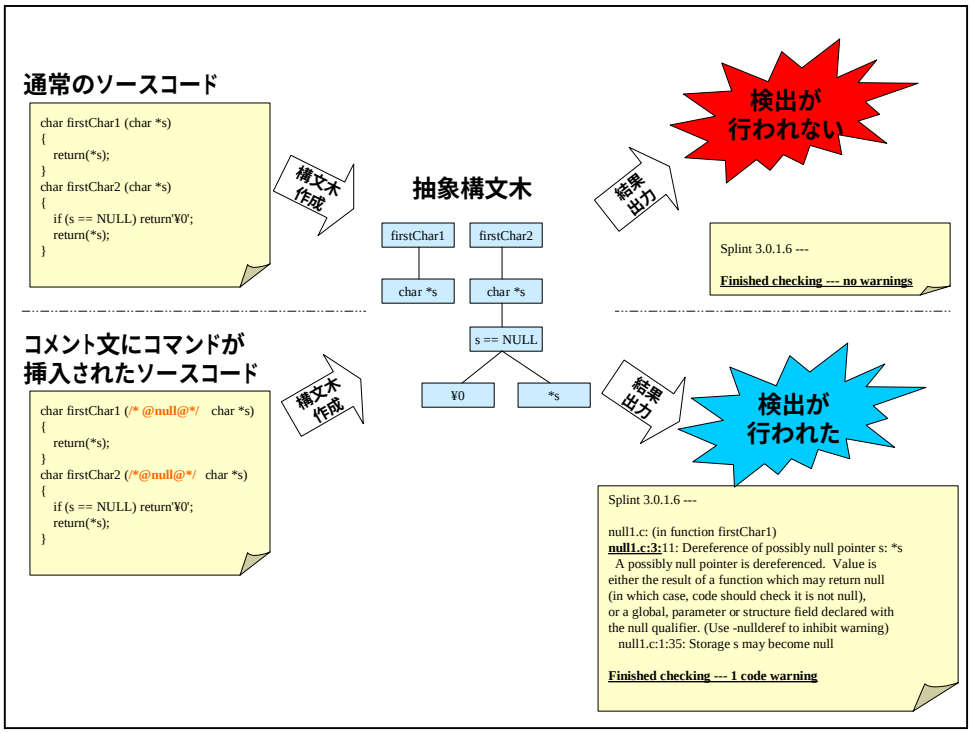

図7. Splint の動作

結果の出力

Splint は、解析結果を標準出力へ出力する。出力は txt 形式でのみ行われる。出力に

含まれる情報は、以下のような項目についてである。

- ファイル名
- ファイルの内容の情報（main 関数の有無等）
- 問題個所の発生個所（何行目に発生したか・先頭から何文字目に発生したか）
- 各行での問題点
- 問題点に対するヒント

開発環境との連携

開発環境と連携を行う機能は備えていない。

D） 開発体制

Splint は、Virginia 大学の Secure Programming Group のメンバーによって、開発・メンテナンスが行われている。David Evans がプロジェクト・リーダであり、主な開発も担当している。メモリの範囲チェックは David Larochelle が開発を行っている。Chris Barker、David Friedman、Mike Lanouette、Hien Phan が David Evans の下で、その他主要個所の開発を担当している。

Splint は NASA Langley Research Center の NSF CAREER Award と NSF CCLI Award、そして MicroSoft 社の援助を受けている。また、開発者である David Larochelle は USENIX student research grant の資金援助を受けている。

Splint の前身であった LCLint は DEC の System Research Center と MIT との間で共同開発されていた。Splint と同様に David Evans が主な設計と担当していた。LCLint は開発の過程で ARPA と NSF と DEC ERP の資金援助を受けており、開発担当者であった David Evans は Intel Foundation Fellowship の援助を受けていた。

E） 現在のバージョンと対応プラットフォーム

対応プラットフォームは UNIX 系 OS だけに留まらず、Windows や OS/2 といったパーソナルユース向けの OS にも対応している。現在の最新バージョンは 3.0.1.6 である。ソースコードの入手は以下のアドレスからできる。

http://www.splint.org/downloads/splint-3.0.1.6.src.tgz

F） 利点と欠点

lint の機能を受け継いだ構文解析機能があり、コメント文にコマンドを記述することで詳細な検査を行えることが特徴である。コマンドは多種多様な記述が可能であり、ドキュメントも詳細に整備されている。ソースコードのコメント文に検査項目を記述するという方式は新しい試みであり、今後これを如何に平易に記述できるようにするかが課題である。

一方で、コメント文に特別な記述を行わずとも十分に検査は行える。これは、標準的なライブラリに関する記述が Splint に十分に用意されているためである。この記述を利用することで、困難になりがちな構文解析型ツールによる検査を、比較的容易に行える点は有用である。

G） その他（対応言語）

ANSI C の C 言語にのみ対応している。C++には未対応である。

2.2.2. Cqual に関する調査

A） 概要

Cqual は米国 California 大学で開発が行われているソースコード検査ツールである。データ型の解析を行うことでソースコード中のデータ遷移の矛盾を導き出す方法が実装されている。このデータ型の解析のために、型修飾子（type qualifier）と呼ばれる C 言語のデータ型の拡張が用意されている。型修飾子の状態遷移が、予めに規定されたルールの範囲内で行われているかを解析することで、データ遷移の矛盾点を特定している。型修飾子はユーザによって独自に定義することが可能である。

検査に先立ってユーザは、ソースコード中のセキュリティ上重要となる個所に型修飾子を記述する。Cqual は起動されると、記述された型修飾子間に矛盾がないかを解析する。矛盾点が発見された時には、Cqual はその矛盾が生じた個所と、矛盾が生じるまでのデータの遷移過程を示す。この遷移過程をユーザが理解しやすくするための対応が行われており、Emacs 上で Cqual の利用を可能とする Program Analysis Mode（PAM）が用意されている。PAM によって、IDE 上での変数・関数の状態遷移を追跡することが可能となっている。

B） 検出可能な脆弱性

Cqual において検査の対象となるのは、主に以下の項目についてである。

(1)ユーザの定義した重要度の高い変数における論理的矛盾

(2)危険な型変換

C） 機能詳細

Cqual はデータ型に注目したソースコード解析ツールである。特徴的なのは、C 言語のデータ型の拡張を行っていることである。この拡張されたデータ型は、型修飾子と呼ばれており、ユーザ（特に、ソースコードの改良を行うプログラマ）によって定義することが可能となっている。ユーザは、この型修飾子をソースコードに挿入することによって、詳細な解析を行うことが可能である。Cqual は型修飾子が挿入されたデータの遷移が正しいかを判別する qualifier interface というインタフェースを有しており、これによってユーザは検査結果の追跡過程を見ることが可能である。以下では、Cqual の機能を次の６つの観点で解説する。

・データ入力
・ソースコードの検査
・型修飾子
・ソースコードの制限
・開発環境との連携
・Web ベースでの利用

データ入力

ユーザが Cqual に引き渡すことができるデータは、検査対象となるソースコードと型修飾子の大小関係に関するルールファイルの２種類である。

①検査対象となるソースコードの読み込み

Cqual は起動時にソースコードの読み込みを行う。後述のように、Cqual が読み込むソースコードは、マクロ展開が行われており K&R 形式で記述されていることが必須となっている。

②ルールファイルの読み込み

ユーザは型修飾子の大小関係のルールを独自に記述することが可能である。ルールには二つの記述が存在し、一つは型修飾子の大小関係、もう一つはライブラリ関数の引数・戻り値における型修飾子の記述である。Cqual では、これらの２つの記述を別々のファイルに行い、それぞれ独立に読み込むことが可能である。

ソースコード検査

検査は主にデータ型と型修飾子の整合性の解析を行う。データ型の整合性の検査は、多くの構文解析ツールで行われるデータ型の整合性の検査と同様である。

①データ型の解析

Cqual は抽象構文木を各データのフローに従って作成する。この構文木からデータ遷移の矛盾を導き出す。また、この構文木の解析では、Cqual は型修飾を必要としていない。従って、ソースコードに型修飾子を記述せずとも、解析を行うことが可能である。

②型修飾子に基づく分析

読み込まれた型修飾子のルールの記述に基づいてソースコード中の型修飾子の遷移を解析する。ここでは、①と同様に抽象構文木を作成し、構文木内での型修飾子の関係を捉えることで、検査を実現している。解析の結果、データがルールの記述外の遷移をした場合には、構文木を基に警告の出力を行う。

以下に Cqual の検査の概要をまとめた図を示す。

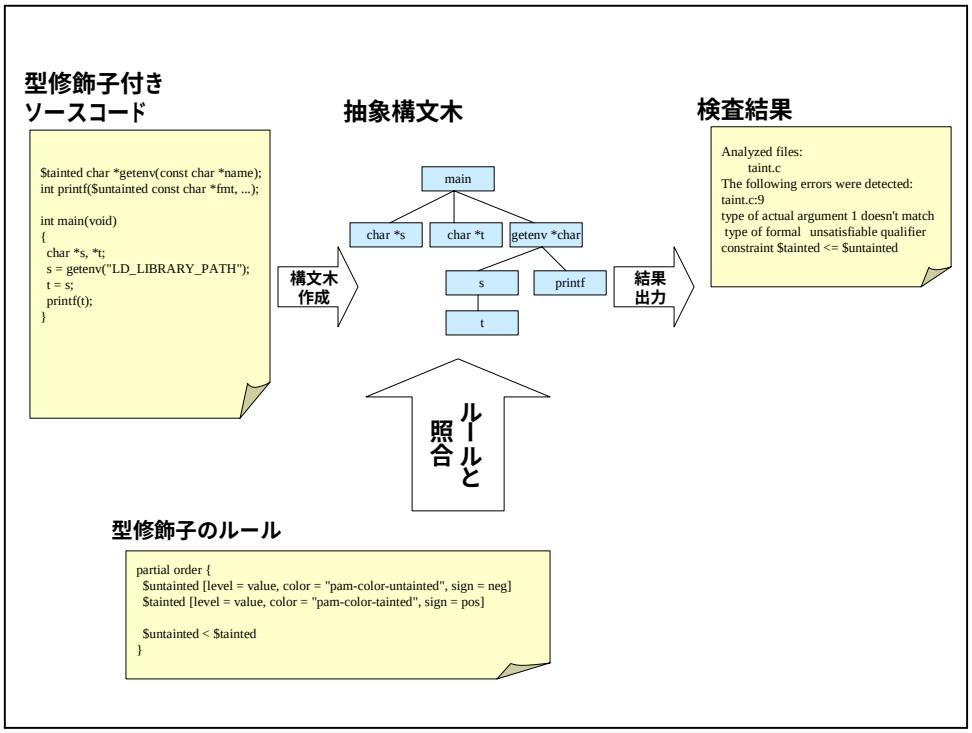

図8. Cqual の動作

型修飾子

Cqual では、型拡張子の作成を行い、型修飾子の遷移を詳細に渡って検査を行うことでソースコード中の誤りを検出する。型修飾子の遷移過程で、定義されたルールと矛盾が生じると、矛盾個所を脆弱性の可能性のある個所としてユーザに警告する。これは、今回調査を行った他の検査ツールでは見られない特徴である。

以下では、型修飾子について解説をする。ここでは例として$tainted、$untainted を使う。型修飾子の持つ意味は以下のようである。

拡張データ型	意味
$tainted	不正の可能性のある変数・定数
$untainted	安全な変数・定数

Cqual では型修飾子に対して、「大小関係」が定義可能である。上記の$tainted、$untainted を以下のように定義する。

```
$untainted  <  $tainted
```

この大小関係の定義から、データの代入時に生じる型修飾子の変換の優先度が定義される。優先度が定義されたことにより、型修飾子の変換が“正しく行われた”または“不正に行われた”と判断することが可能になる。

この優先度の定義による解析の考え方を、以下のソースコードで解説する。

```
1  $tainted char *getenv(const char *name);
2  int printf($untainted const char *fmt, ...);
3
4  int main(void)
5  {
6      char *s, *t;
7      s = getenv("LD_LIBRARY_PATH");
8      t = s;
9      printf(t);
10 }
```

上記のソースコードにおける型修飾子の遷移は下記のようになる。

行	解説
1 行目	getenv 関数の宣言を行っている。この関数の返り値は、$tainted によって「常に安全でない」と宣言されている。
2 行目	printf 関数の宣言を行っている。この関数の第一引数として渡される変数は常に安全であると宣言されている。
7 行目	変数 s は getenv 関数の返り値であることから、$tainted と解釈される。
8 行目	変数 t は変数 s を代入したことから、変数 t は$tainted と解釈される。
9 行目	引数として変数 t を代入している。しかし、2 行目の宣言によると、第一引数は安全な変数を代入すると定めていることから、安全な変数が代入されるという宣言と矛盾が生じている。

上記ソースコードを Cqual で検査を行うと、以下のようなメッセージが出力される。

```
Analyzed files:
        taint.c
The following errors were detected:
taint.c:9
type of actual argument 1 doesn't match type of formal
unsatisfiable qualifier constraint $tainted <= $untainted
```

検査結果では、9 行目で矛盾が生じていることが指摘されている。前述した例でのソースコードの 1、2 行目を以下のように変更する。

```
1  $untainted char *getenv(const char *name);
2  int printf($tainted const char *fmt, ...);
```

上記の変更を行ったソースコードの検査を行うと、Cqual からの警告メッセージは出力されない。1、2 行目の宣言が変更されたことにより、9 行目の printf 関数の第一引数に渡される変数 t は、$untainted と判断される。一方で ptintf 関数の宣言では第一引数で渡される変数は、$tainted となっている。この結果、Cqual は警告メッセージの出力を行うはずだが、実際には警告メッセージの出力は行われない。これは、Cqual は定義された大小関係に従って、ソースコードの分析を行っているためである。

ソースコードの制限

Cqual は、検査を行うソースコードに以下の制限を行っている。

①K&R 方式で記述されたソースコードの排除

Cqual は ANSI C に準拠した関数宣言以外の宣言は解析を行わない仕様となっている。K&R 形式で宣言された関数を有するソースコードを Cqual で検査すると、エラーメッセージが出力され、検査が停止される。従って、ユーザは事前に K&R 形式で記述された個所を ANSI 形式に変更する必要が生じる。これには、GNU パッケージの一つである protoize というツールが有効であり、protoize によって関数の定義・宣言を機械的に変更することが可能になる。但し、protoize による変換は、元のソースコード中に記載された論理的な内容を維持する保証はなされていないので、注意が必要である。

②マクロ記述

マクロで記述された個所の誤検査の防止・検査漏れの防止を行うことは、ソースコード検査ツールにとっては重要な意味を持つ。マクロ記述によって、柔軟なソースコードの作成が可能になった反面、脆弱性のあるプログラムを作成する可能性も高くなった。

このような背景から Cqual では、検査過程におけるマクロ記述のされたソースコードの検査を行わない仕様になっている。この為、ユーザはマクロ展開が行われたソースコードに対して検査を行うことになる。ソースコードにマクロ宣言が行われていると Cqual はワーニングメッセージを出力する。また、マクロ宣言以外にもコメント文が挿入されていると、ワーニングメッセージを出力する。

マニュアルにおいてマクロの取り扱いについて言及されている。マクロを展開したソースコードを生成する方法もマニュアルで紹介されており、それは次のような方法である。

・gcc のコマンドオプション”-E”によってプリプロセス処理が終了したコードを生成する。

・コードを別名でファイルに保存し、そのファイルに対して Cqual で検査を行う。

これによって生成されたコードの検査を行った時、Cqual は大元のソースコードにおける行番号を沿えてメッセージを出力する。また、一括でマクロ展開をしたファイルの作成を行うには、以下のような記述を Makefile に行うことで可能となる。

```
.c.o:
$(CC) -E $< | remblanks > $*.ii
perl remquals < $*.ii > $*.i
$(CC) $(CFLAGS) -c -o $*.o $*.i
mv -f $*.ii $*.i
```

尚、上記中の “ remblanks ”、“ remquals ” は Cqual の展開フォルダの bin ディレクトリ以下に置かれている。

Web ベースでの利用

Web 上で Cqual を利用することができる。利用は以下のアドレスから行うことができる。

http://lagaffe.cs.berkeley.edu/

ユーザは検査を行いたいソースファイルを CGI 経由でサーバに転送し、サーバで行われた検査結果を Web ブラウザ上で確認することが可能である。これにより、Cqual を導入することが困難な環境でも、ソースコード検査を行うことが可能である。

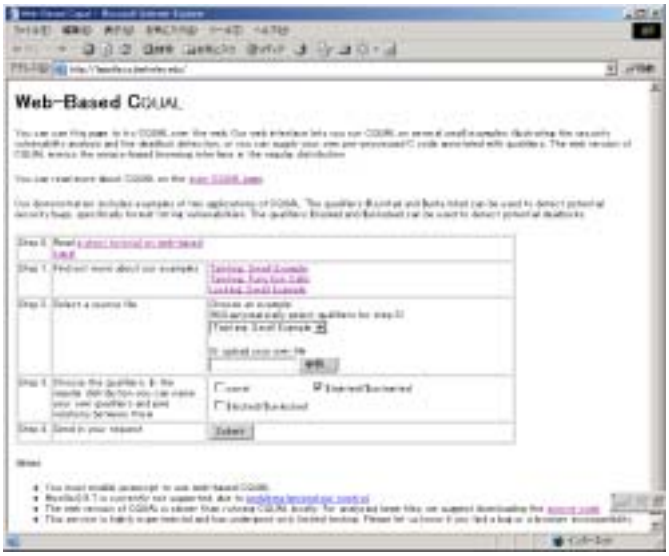

図9. Web-Base Cqual 起動画面

Web ベースの検査では、ユーザがローカル保有しているソースコードの検査の他に、予め用意された sample コードに対する検査の２種類ソースコードの検査が可能である。尚、ローカルに保有しているソースコードの検査は一度に一つのソースコードに対してしか行えない。検査を行いたい型修飾子について選択可能であり、const、$tainted / $untainted、$locked / $unlocked の三種類が選択可能である。ユーザ独自のソースコードで検査を行うには、これらの型修飾子をソースコードに記述することになる。Web ベースでの検査結果は以下のようになる。

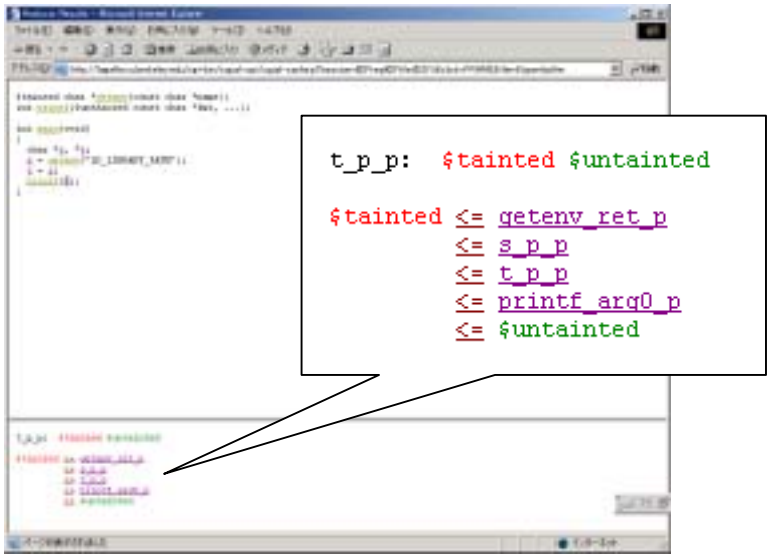

図10. Cqual の解析結果の画面

Web ベースでの検査結果は上図のように色分けされて表示されている。色分けは型修飾子毎に行われている。下線付きの変数、不等号記号にはそれぞれリンクが張られており、リンク先では型修飾子同士の比較の過程が表示される。この機能を用いて、型修飾子の遷移を追跡することで、問題発生個所が特定可能である。

開発環境との連携

Cqual は、IDE による操作環境として Program Analysis Mode（PAM）が用意されている。これによって Emacs 上で Cqual が利用可能である。起動するには、Emacs 上で“ M-x cqual ”と入力する。

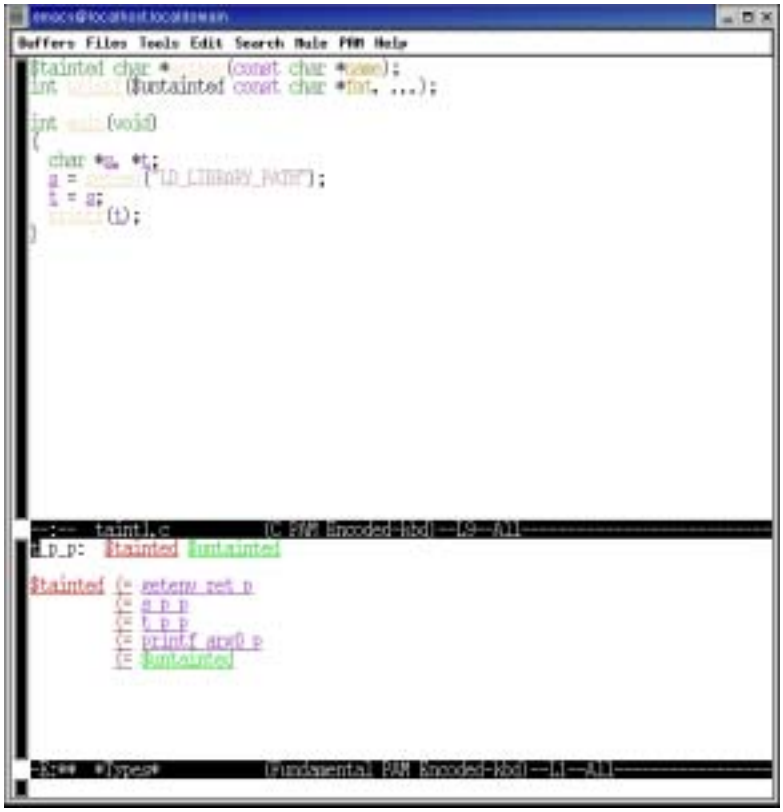

図11.　PAM による検査結果画面

上図は PAM で検査を行った画面である。Emacs 上で Cqual を動かした場合、Web 上で動作させた場合と同様の操作が可能である。また、Emacs 上での操作においてポインティングデバイスを利用できない環境は、コマンドを使うことでポインティングデバイスを利用しているのと、同様の操作が可能である。

D）　開発体制

開発は Jeff Foster（California 大学 Berkeley 校）の手によって行われている。Cqual の開発は The Open Source Quality Project (OSQ)の一部である。OSQ プロジェクトでは、オープンソース・ソフトウェアの質向上の為の設計・開発の補助ツールを開発している。OSQ プロジェクトの成果物のライセンスは GPL となっており、Cqual も GPL である。

E）　現在のバージョンと対応プラットフォーム

Cqual は現在の最新バージョンは、0.98 である。ライセンスは GPL となっている。Web 上での検査も行えることから、Web ブラウザが利用可能なプラットフォームであれば、広く利用が行える。Web ベースでの検査でも、検査を行えるのは ANSI C に準拠した範囲に対してのみである。ローカルな環境へのインストールは、UNIX 系の環境であれ

ば可能である。また、PAM を利用する場合には Emacs が導入されている必要がある。

F） 利点と欠点

Cqual はデータ型の遷移を重点的に解析を行うという点で他のツールには見られない特徴がある。検査を行いたいデータ宣言に型拡張子を挿入することで、脆弱性の可能性のあるデータ遷移の解析結果を得ることができ、厳密な解析を行う必要がある場合には有効である。一方で、ソースコード中に型拡張子の記述をしなければならず、ソースコードに対する作業が発生するという点が欠点となる。

標準のルールファイルには、幾つかのライブラリ関数やデータ型における大小関係の記述が行われているので、これを利用することで検査を行うことは可能である。しかし、型修飾子をソースコードに記述しない状態での検査を行うった場合、有益な情報を得られることは殆どない。この点には注意が必要である。拡張データ型はユーザ独自に作成を行うことができるが、これは標準のルール定義との整合性の維持が難しく、アプリケーションの設計の理解を行わずに検査を行うのは困難である。

同様な構文解析型のツールである Splint もデータ遷移の解析を行う。Splint のデータ遷移の解析は、全てのデータに対して行い、特定のデータに対しての解析は行わない。Cqual では型拡張子を用いることで、特定のデータに対しての解析を行うことが可能である。このことから、Cqual は重点的に検査を行う検査対象が決まっている場合には有用である。

G） 対応言語

ANSI C の条件を満たす C 言語にのみ対応している。C++言語に対しては未対応である。

3. ソースコード検査技術の比較

3.1. 検査精度及び検査手順

ここでは2章で取り上げた各技術によって、実際に CERT/CC 等でこれまでに脆弱性を持つと指摘されたオープンソース・ソフトウェアの脆弱性を事前に検出することが可能であるかを検証する。1．3．1 で説明をした各アプリケーションに対して、各技術を適用し、脆弱性に関してどのような検査結果が得られるか、または得られないかを調査した。

3.1.1. 実験環境

実験環境の選択にあたっては、以下の 2 点を考慮し、下記に示す環境を設定した。

1. 1.3.1 章で取り上げたすべての技術を同一の環境で評価する
2. 対象となる脆弱性を持つアプリケーションが実効可能である

CPU	Celeron 534MHz
キャッシュメモリ	128 KB
メインメモリ	128 MB
OS	Red Hat Linux 6.2 kernel 2.2.14
その他	gcc2.91.66 で検査ツールのコンパイルを行った。

図12. 実験環境

3.1.2. 脆弱性を持つアプリケーション

各技術の効果を網羅的に調査するために、ここでは以下の脆弱性を持つアプリケーションを対象とした。すべてのアプリケーションにおいて脆弱性をもつバージョンを取得し、実験環境にインストールし、実際のアタックコードを作成して、攻撃を行った。

Bind: Bind は最も有名なドメイン・ネーム・システムで Internet Software Consortium(ISC)によって提供されている。Bind8.1.1 というバージョンには、Bind を実行しているユーザの権限で任意のコードを実行可能なリモート攻撃の脆弱性が存在する。（一般的には特権が奪われる）この脆弱性は一般的に ”Bind iquery Buffer Overflow” と呼ばれている。この脆弱性では、Bind は iquery パケットを受け取ると、そのパケットの回答部に格納されたアドレスを内部バッファに格納して要求に答えようとする。しかし、この内部バッファへのコピーの際、コピー元のデータサイズのチェックが行われず、受信したパケットの回答部に格納されたデータのサイズが、内部バッファサイズより大きい場合は、内部バッファのメモリ領域を越えて関係のないメモリ領域を上書きされる。内部バッファはスタックに確保されているため、典型的なバッファーオーバーフローが発生する。実際には以下のよう

に memcpy()関数に起因した脆弱性が存在する。

```
1       ns_debug(ns_log_default, 1,
2               "req: lquery class %d type %d", class, type);
3       fname = (char *)msg+HFIXEDZ;
4       alen = (char *)*cpp - fname;
5       memcpy(anbuf, fname, alen);
6       data = anbuf + alen - dlen;
7       *cpp = (u_char *)fname;
8       *buflenp -= HFIXEDSZ;
9       count = 0;
```

図13. req_iquery()内の脆弱性

qpopper: qpopper は Qualcomm が提供している POP (Post Office Protocol) サーバを構築するときに利用するソフトウェアである。qpopper の 2.4 というバージョンにはバッファーオーバーフローに関する脆弱性が存在している。qpopper はルート権限で動作するために、バッファーオーバーフローが発生するとルートの権限が奪取される。Qpopper ではメッセージを表示するルーチンに脆弱性が存在する。クライアント側から送信されたデータを表示する際に、ある長さを超えたメッセージの場合にはバッファが溢れ、オーバーフローが発生する。qpopper の脆弱性は以下のように vsprintf()関数の利用に起因して発生する。

```
1       #ifdef HAVE_VSPRINTF
2               vsprintf(mp, format, ap);
3       #else
4       #ifdef PYRAMID
5               (void)sprintf(mp, format, arg1, arg2, arg3, arg4, arg5, arg6);
6       #else
6               (void)sprintf(mp, format, ((int *)ap)[0], ((int *)ap)[1],
7               ((int *)ap)[2], ((int *)ap[3], ((int *)ap)[4]);
8       #endif
9       #endif
```

図14. pop_msg.c ファイル内の脆弱性

elm: elm は linux の mail コマンドである。elm の 2.5.0 というバージョンには、バッファーオーバーフローに関連する脆弱性が存在している。elm は setgid されたプログラムであるために、バッファーオーバーフロー攻撃によってルート権限が奪取される。elm の脆弱性はコード中の strcpy() 関数の利用に起因している。

```
1       char termname[40];
2       char *strcpy(), *getenv();
3       if (getenv("TERM") == NULL) return(-1);
4       if (strcpy(termname, getenv("TERM")) == NULL)
5               return(-1);
```

図15.　curses.c ファイル内の脆弱性

imap: imap (Internet Message Access Protocol) は POP と同様にメールの送受信・配信を制御するプロトコルである。POP との違いは、POP はメールサーバにあるすべてのメールをクライアント側に取り込むのに対して、imap はメールサーバにあるメール一覧をもらい、読みたいメールだけ受信することができる点である。imap4.0 には、バッファーオーバーフローに関連する脆弱性が存在している。この脆弱性をついて、不正に任意のコマンドをリモートから実行することが可能である。以下に示すように、strcpy() 関数において、スタック上の固定サイズのバッファーオーバーフローさせる。

```
1       long server_login(char *user, char *pass, int argc, char *argv[])
2       {
3               char tmp[MAILTMPLEN];
4               struct passwd *pw = getpwnam(user);
5                       /* allow case-independent match */
6               if (!pw) pw = getpwnam(lcase (strcpy(tmp, user)));
7               if (!(pw && pw->pw_uid &&       /* validate user and passwd */
8                 !strcmp(pw->pw_passwd, (char *)crypt(pass, pw->pw_passwd)))
9               return NIL;
```

図16.　log_std.c ファイル内の脆弱性

apache: apache は最も有名な Web サーバである。Apache1.1.3 というバージョンにはシンボリックリンク攻撃の脆弱性がある。Apache では apache_status というファイルを/tmp の下に作成する。作成にあたってファイルの存在をチェックしないため、たとえば、apache_status が/etc/passwd にシンボリックリンクされていた場合には、/etc/passwd のアクセス権限を変更してしまう。以下に示す個所で上記の問題が発生する。(通常のファイルでは問題にならないが/tmp の時に問題となる場合がある。)

```
1       API_EXPORT(int) ap_popenf(pool *a, const char *name, int flg, int mode)
2       {
3           int fd;
4           int save_errno;
5
6           ap_block_alarms();
7           fd = open(name, flg, mode);
8           save_errno = errno;
9           if (fd >= 0) {
10              fd = ap_slack(fd, AP_SLACK_HIGH);
11              ap_note_cleanups_for_fd(a, fd);
12          }
13          ap_unblock_alarms();
14          errno = save_errno;
15          return fd;
16      }
```

図17.　alloc.c ファイル内の脆弱性

wu-ftpd: wu-ftpd はワシントン大学で開発された FTP サーバである。高度なセキュリティ機能をもつことで知られ、幅広く利用されている。wu-ftpd2.6.0 というバージョンには、フォーマットストリング攻撃に対する脆弱性が存在する。この脆弱性によりリモートまたはローカルなユーザは特権で任意のコマンドを実行することが可能となる。この脆弱性は以下に示すように setproctitle() 関数内の vsprintf() 関数に起因して発生する。

```
1       /* print ftpd: heading for grep */
2       (void) strcpy(p, "ftpd: ");
3       p += strlen(p);
4
5       /* print the argument string */
6       VA_START(fmt);
7       (void) vsnprintf(p, SPACELEFT(buf, p), fmt, ap);
8       VA_END;
```

図18.　ftpd.c ファイル内の脆弱性

3.1.3. 検査方針

検査実験は、大きく分けて下記の３つの観点で行う。

① 検査ツールが既知の脆弱性を検出するかを確かめる。
② 検査速度の比較を行う。
③ 検査ツールによる検出がどの程度困難なのかを確かめる。

また、実験にあたっては検査ツールを駆使しての検査は行わない。検査ツールの知識を有していないユーザが検査を実施することを想定して検査を行う。

3.1.4. 検査ツールに関する留意点と予測される結果

予想される結果

２章では検査ツールを大きく「パターンマッチング型」と「構文解析型」の２つに分類を行っていた。パターンマッチング型の検査ツールでは、使用されている危険度の高い関数を全て列挙する為、脆弱性を含む・含まないに関わらず可能性がある全ての個所について警告が行われる。このことから脆弱性に関する情報が含まれる可能性が高い反面、それ以外の検出結果も多量に得る可能性が高いと予測される。一方で、構文解析型の検査ツールでは、精密な検査を行うためにソースコードに解析の構文を記述する必要が生じる。しかし、今回の検査テストでは、ソースコードを極力改変しないという方針になっている。このことから、脆弱性を検出できない可能性が高い反面、検出が行われた場合には、比較的容易に脆弱性の個所を見つけ出せることが予想される。

検査ツールの留意点

①パターンマッチング型の留意点

ソースコードの検査について、特別な留意点はない。

②構文解析型の留意点

Splint

ヘッダファイルの検索もツールが行うが、検索がうまくいかない場合、検査を中止されてしまうことがある。

Cqual

ソースコードにマクロやコメントの記述がある場合には、正常な検査が行えない場合がある。

3.2. 検査結果

1.3.2. で示した５つの既知の脆弱性を持つツールについて検査を行った。

3.2.1. 検出確認

既知の脆弱性がソースコードから検出が行えるかを実験する。検出確認は、3.1.3.の方針に従って下記の条件で行うこととする。

・ 既知の脆弱性を含むファイルに対してのみ検査を行う。未知の脆弱性については、検査の対象としない。
・ 脆弱性を含む個所についての警告が得られた場合は、脆弱性を検出できたと判定する。但し、脆弱性の警告の内容についても吟味する。吟味の結果、実際の脆弱性と乖離のある警告の場合には、検出が行われなかったと判定する。
・ 検査ツールは可能な限りデフォルトの設定を使う。
・ デフォルトの設定で検出が行えなかった場合のみ、オプション等の適用を試みる。
・ ファイルの配置等の問題で検査ツールの起動に失敗する場合は、適宜対象ファイルの編集を行う。但し、編集を行う個所は必要最小限に留める。

各アプリケーションに対する検知結果は次のようになった。

	RATS	ITS4	Flawfinder	Splint	Cqual
elm	○	○	○	△*2	×*4
qpopper	○	○	○	△*3	×*4
imap	○	○	○	×	×*4
apache	△*1	○	○	×	×*4
bind	△*1	△*1	○	△*2	×*4
wu-ftpd	×	○	○	×*5	×*4

○：正常に検出を行うことができた。

△：検出を行うにあたって注意点があるが、検出は可能である。

×：検出を行うことができなかった。

*1：オプションを利用して、全ての警告を出力することで検出がされた。
*2：別ディレクトリに保存されたヘッダファイルをソースコードと同じディレクトに保存した。
*3：ソースコードが参照するヘッダファイルのリストを一部書き換えた。
*4：マクロの展開・コメント文の削除を行ったファイルに対して検査を行った。
*5：ソースコードの一部を変更した。

検出結果に関する考察

「パターンマッチング型」と「構文解析型」とでは、顕著な差が出た。「パターンマッチング型」では、全ての検査ツールでほぼ全てのアプリケーションに対して検出を行うことができた。

「パターンマッチング型」は、内部データベースの記述に従って、データベースに記述されている関数の使用の有無を検索している。今回検査を行ったアプリケーションの脆弱性を引き起こしているのは、sprintf や strcpy といった脆弱性を引き起こす可能性が高いことで知られた関数であった。従って、これらの関数はデータベースに登録が行われている可能性が高いため、検出に成功した。しかし、脆弱性の存在が知られていない関数やユーザ独自に作成した外部関数に脆弱性が存在する場合は、検出が困難であると考えられる。

「構文解析型」は、プログラムの開始から呼び出される関数や定義される変数の関係を

詳細に渡って追跡を行う。この追跡過程において生じた矛盾点がバグの可能性が高い個所であり。バグの特別な状態が脆弱性に関連している。従って、特定のセキュリティ上の問題を特定するためには、ソースコード中に解析ルールの記述を行う必要がある。今回の実験ではソースコードに改変を加えない方針でテストを行った。結果として、検出成果は芳しくなかった。また、Cqual はソースコード中に拡張データ型を記述することで解析を行うツールのため、全てのアプリケーションにおいて検出に失敗している。また、Splint では、ファイルの保存場所等の変更を行わないと検査を行えなかった。

3.2.2. 速度比較

ユーザが実際にツールを扱う際には、比較的短時間で検査を行えることが望ましい。そこで、各ツールにおける検査時間の測定を行った。ツールによっては、内部に時間計測を行う機能を有するものもある。しかし、これらの時間計測機能は仕様が統一されておらず、一様な測定・比較を行うことが困難だと考えられる。従って、以下のようなルールに従って、各々の検査速度の測定を行った。

- 速度計測は time コマンドに対して適用した。
- time コマンドにおける user 時間と system 時間の和を計測時間とする。
- swap 等を避けるために、実験機では必要最小限のプロセスしか起動しない。
- 各ツールにおいての計測は１０回行い、その平均値をデータとした。

	RATS	ITS4	Flawfinder	Splint	Cqual
elm	0.028	0.018	0.483	0.262	0.014
qpopper	0.026	0.01	0.21	0.126	0.005
imap	0.026	0.009	0.203	0.046	0.005
apache	0.03	0.02	0.546	0.487	0.018
bind	0.032	0.026	0.908	0.837	0.033
wu-ftpd	0.068	0.127	3.781	1.511	0.338

速度比較に関する考察

「パターンマッチング型」の RATS や ITS4 が比較的高速な結果となった。Flawfinder の速度が高速とは言い難い結果となったのは、Flawfinder の実装は Python スクリプトで行われているためであろう。設計において、RATS や ITS4 と比べて特別な差異が見られない。このことからも Flawfinder の速度の問題は、Python 上の問題であると考えられる。

一方で、「構文解析型」では速度差が顕著であった。Cqual が高速であるという計測結果が得られた。しかし、Cqual による検査では、①でも述べたように、ソースコード中に拡張データ型の記述を行い、そのデータ型同士の論理的不整合から脆弱性を割り出す。今回のテストにおいては、ソースコードに特別な記述は加えていない。従って、ユーザがソースコードに対して行って欲しい解析は行われていない。この為、速度が高速であるとい

う計測結果で出たと推測される。一方で Splint では、ファイルの保存場所の変更を行ったために、標準の構文解析機能をベースとした検査を行うことができた。構文解析を行った状態でも、Flawfinder より早い結果が得られている。

3.2.3. 検査の容易さ

ここでは、一般的な開発者以外のユーザが、検査の容易さに視点を置く。容易さの尺度として、以下の２点に着目したい。

- 検出結果の中から脆弱性と関係のある個所をユーザが見つけやすいか
- 見つけにくい場合には条件を絞った検査を行うことができるか

検査結果の個数

検査結果として出力された警告の個数に着目する。検出を行いたいファイルに対して、警告の個数が検査ツール毎にどの程度違いが出るかを下記のように表にまとめた。

	RATS	ITS4	Flawfinder	Splint	Cqual
elm	21	13	22	181	4
qpopper	8	5	10	23	12
imap	4	9	5	37	22
apache	16	2	12	178	15
bind	29	18	30	272	10
wu-ftpd	379	332	303	946	70
総計	457	379	382	703	133

検査結果の出力

今回の検査では、各ツールからの出力を全て txt 形式で行うようにした。txt 形式で出力される検出結果の警告文の出力形式を、次のような方針で整理を行った。

- 警告の出力順（危険度順、脆弱性の種類順、行番号順）
  分類方法を２つ採用されている場合は、次のように表記する。
  例：危険度順→種類順
- 脆弱性の解説文と対策文の有無

	RATS	ITS4	Flawfinder	Splint	Cqual
出力順	種類順→ 行番号順	種類順→ 行番号順	危険度順→ 行番号順	行番号順	行番号順
解説文 の有無	解説文：○ 対策文：○	解説文：○ 対策文：○	解説文：○ 対策文：○	解説文：○ 対策文：○	解説文：○ 対策文：×

特定の関数に対する検査

今回の検査では特別なオプションを用いずに検査を行った。ここでは、コマンドのツールにオプションで関数名を引き渡すことでその関数の使用状況の検査が行えるかを下記の表にまとめた。

	RATS	ITS4	Flawfinder	Splint	Cqual
関数の検査	○	○	×	×	×

検査の容易さに関する考察

検査結果の個数について以下に述べる。「パターンマッチング型」では、RATS での検査結果の個数が比較的多い。これは、RATS データベースの登録数の差から生じているとも考えられる。「構文解析型」では、Splint の個数が全ての検査ツールと比較しても多い。検査結果の出力の中には“型の不一致”や“戻り値の代入先のない関数の使用”等、細かい警告まで出力されている。この結果、Splint での検査結果は詳細な情報を入手できるが、逆に必要な情報を見つけ出すのが困難であるとも言える。また、①の検査結果とあわせて考察すると検査結果の数と比較して、検出が行えた結果が少ない。Cqual の出力結果はデータ型に対する警告が殆どであり、脆弱性に対する問題等の指摘は殆ど見受けられなかった。

検査結果の出力については、「パターンマッチング型」のツール群では、解説文・対策文が充実している。一方で、「構文解析型」のツール群では、解析結果として得られた矛盾点を指摘するのみに留まっている。このことから、ユーザは矛盾点が生じた理由を考察しなければならない。また、出力形式に関しては、各ツールでそれぞれの特徴がある。今回のように開発者でないユーザを対象とした場合は、危険度順に表示をされる方が望ましい。その点では、標準で危険度順にデータの表示を行う Flawfinder がユーザには分かりやすいと思われる。

特定の関数に関する検査について述べる。実際の検査においてユーザが全ての情報を把握することは現実的ではない。そこで、危険度の高い関数を数個について調べたい場合が考えられる。そのような使い方に対応しているのは、RATS と ITS4 である。「構文解析型」のツールは、その仕様から関数名をベースにした検査を行うことができない。

3.3. 総括

3.3.1. RATS

RATS は、今回の実験の中では際立った特長が結果に表れなかったツールである。このことは逆に、汎用的に使用できるツールであるとも考えられる。今回の検査では、脆弱性を標準で検出を行えたアプリケーションが 3 つ、RATS のオプションを利用することで検出できたアプリケーションが 2 つ、検出が行えなかったアプリケーションが 1 つという結果が出た。検出が行えなかった場合があるものの、テスト結果は実用に耐えうるものだと考えられる。検査速度も比較的早い結果となった。検査結果の出力では、警告文を複数人の手で作成していることもあり、充実した内容になっている。一方で、検査結果の出力数は比較的多い結果になっている。これは、RATS のデータベースに登録してある検査項目

が充実しているということが要因となっていると考えられる。この点においては、ユーザは検査結果の分析時に留意する必要があると考えられる。

全体的にバランスの取れたツールであり、アプリケーションの脆弱性の可能性を把握したユーザには有効であると考えられる。一方で、精密な検査を行いたいユーザやアプリケーションの開発者は、別の際立った特長のあるツールの使用を視野に入れる必要がある。

3.3.2. ITS4

ITS4 は今回の実験の中では、最も高速な動作をしたツールである。検出結果も標準で検出できたのが 5 つ、ITS４のオプションを利用することで検出ができたのが 1 つという結果になっている。条件付きのケースが一つ存在したが、全ての脆弱性を検出することができた。出力結果の個数も今回実験を行ったツールの中では平均的な個数であった。また、検査を行いたい関数名をキーに検査を行う機能も有しており、必要な情報をユーザの判断で取り出すことが可能となっている。一般的なユーザがアプリケーションの検査を行うにあたって必要な機能は十分に備えていると言える。

但し、ITS4 の使用には注意を要する。「非商用利用に限って」GPL ライセンスとして配布されていることである。この為、ユーザはライセンスの確認を十分に行ってから使用を行うことになる。

3.3.3. Flawfinder

Flawfinder は、今回の実験で、全てのアプリケーションの脆弱性を標準で検出することができた唯一の検査ツールである。検査結果の表示が危険度順に表記されるように設計されており、開発者以外のユーザが使用することを考慮されている。警告の出力数も今回実験を行ったツールの中では平均的な個数であり、必要な情報を見落とす可能性は比較的少ないと考えられる。有用な検査結果を得るという観点では、Flawfinder は優れたツールであると言える。しかし、Python スクリプトで実装を行われているため、検査速度は非常に遅い。今回は一つのファイルに対してのみ、検査を行った。これをアプリケーションに含まれる全てのファイルに対して検査を行う場合には、多大な時間を要する可能性がある。また、Flawfinder が所有するデータベースは Python スクリプトの中に内包されており、ユーザが独自の検査ルールを作成するのは容易ではない。

Flawfinder は検査を行うか行わないかという観点では非常に優れているが、実際にユーザが使用する際の柔軟性と速度には留意が必要である。

3.3.4. Splint

Splint は、構文解析ツールがベースとなって開発された検査ツールである。この為、検査を行うソースコード以外に、ソースコードに記述されている関数の定義等が記載されているファイルに対しての検査も行う。今回の実験は、アプリケーションに対して特別な変

更を行わずに検査を実行するという方針であった。しかし、先述した仕様の為、今回の方針に従うと検査を行うことができなかった。従って、ソースコードとヘッダファイルが同一ディレクトリに保存されていない場合には、保存場所を統一する等の作業を行った。これはファイルの配置等を動かすだけであり、困難な作業ではない。しかし、BIND のようなソースコードの配置が複雑なアプリケーションでは、比較的注意を要することになる。

実際の検査においては、Splint は多種多様な警告の出力をした。警告の出力数で比較をすると、今回実験を行った 5 つの検査ツールの中で最も多かった。しかし、出力数が多いということが、脆弱性発見の手がかりになるとは限らない。これは今回の実験で Splint が脆弱性を検出できたのが半数のアプリケーションに対してのみであったということから分かる。また、警告の内容を調べてみると変数宣言の問題等、直接的に脆弱性に関係しない警告が多いことが分かる。

今回の実験は、ソースコードを改変しないという方針であった。一方で、Splint では精密な検査を行うためには、ソースコードに Splint のルールを記述する必要がある。この為、精密な検査を行うことができなかった。ルールの記述が正確に行われているならば、比較的正確な検査が行える。ルールの記述を正しく行える且つ、出力される警告の分析を行えるユーザや開発者にとっては、有効なツールであると考えられる。

3.3.5. Cqual

Cqual は C 言語のデータ型を拡張し、その拡張データ型がプログラムの中でどのように変遷していくかを解析するツールである。このツールの特性上、検査を行うソースコードの改変を行うことが望ましい。しかし、今回の実験では、ソースコードの改変を行うという作業はユーザにとっては現実的な作業ではないという観点で改変を行わなかった。この為、全てのアプリケーションで脆弱性の検出は確認できなかった。また、Cqual では、「K&R 形式で記述されたソースコードは検査を行わない」・「マクロは展開されていないと検査は行わない」という仕様になっている為に、検査を行うには、ソースコードに対して加工をする必要がある。この点が実際の検査においては障害になると考えられる。また、ソースコードに Cqual のルールを書き込むには、データ定義等のアプリケーションの設計をある程度理解している必要がある。この点もまた、ユーザにとっては障害になるであろうと考えられる。

4. まとめ

ソースコード検査技術の比較の結果から、開発者以外のユーザがオープンソース・ソフトウェアの検査を行う場合には、パターンマッチング技術のツールの方が利用に適していると言える。構文解析型のツールは、ユーザがソースコードをツールに合わせて加工を行うことが想定されている。この点からも、一般的なユーザを対象とした際には、構文解析型のツールは検査を行うには適していない。しかし、開発者の視点に立って評価を行った場合には、構文解析型のツールも違う評価がなされる可能性がある。何故なら、２つの構文解析型のツールは、ソースコードに各ツールのルールを記述することで、その機能を発揮するからである。解析の精度は非常に高いことから、信頼性の高い検査が行えると考えられる。構文解析ツールのルールの設定方法等は、改良の余地があり開発者間でも議論が続けられている。

パターンマッチング型のツールでは、それぞれ特徴がありユーザが状況に適したツールを選択することが最良であると考えられる。それぞれの３つのツールは、検査能力の高さでは Flawfinder、検査速度の速さでは ITS4、ユーザインタフェースの充実・カスタマイズの容易さでは RATS が勝っている。これらの３つのツールからオープンソース・ソフトウェアの検査に最も適したツールを選択するとすれば、ITS4 と RATS が挙げられる。検査精度の高さ以上に、カスタマイズの容易さが一般的なユーザに適していると考えられることと、ライセンスの制約を考慮すると、広く一般のユーザが利用する場合には RATS が適しているといえる。

参考文献

[1] “Sardonix Security Portal”, https://sardonix.org

[2] Gary McGraw, John Viega, “安全なソフトウェアを作成する：セキュアソフトウェアの保証”, IBM, 2002, http://www-6.ibm.com/jp/developerworks/security/assurance.html

[3] “NCSA Secure Programming Guidelines”, NCSA Servers, http://archieve.ncsa.uiuc.edu/General/Grid/ACES/security/programming/

[4] “Source Code Scanners for Better Code”, Linux Journal, 2002, http://www.linuxjournal.com//article.php?sid=5673

[5] “Secure UNIX Programming FAQ”, 1999, http://www.whitefang.com/sup/secure-faq.html

[6] “How to Write Secure Code”, 2002, http://www.shmoo.com/securecode/

[7] John Viega, Gary McGraw 著, “Building Secure Software”

[8] SecureSoftware Web Page, http://www.securesoftware.com/

[9] John Viega, McGraw, “ITS4 A Static Vulnerability Scanner for C and C++ Code”, Reliable Software Technologies, http://citeseer.nj.nec.com/viega00its.html

[10] “Flawfinder Home Page”, http://www.dwheeler.com/flawfinder/

[11] David A. Wheeler, “Flawfinder manual”, http://www.dwheeler.com/flawfinder/flawfinder.pdf

[12] David A. Wheeler, “Secure Programming for Linux and Unix HOWTO”, 2002, http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html

[13] Splint Home Page, http://www.splint.org

[14] David Evans, David Larochelle, “Improving Security Using Extensible Lightweight Static Analysis”, IEEE Software, 2002, http://www.cs.virginia.edu/~evans/pubs/ieeesoftware-abstract.html

[15] David Larochellle, David Evans, “Statically Detecting Buffer Overflow Vulnerabilities”, USENIX, 2001, http://www.cs.virginia.edu/~evans/usenix01-abstract.html

[16] David Evans, John Guttag, James Horning, Yang Meng Tan, “LCLint: A Tool for Using Specifications to Check Code”, Symposium on the Foundations of Software Engineering, 1994,
http://www.cs.virginia.edu/~evans/sigsoft94.html

[17] CQUAL　A tool for adding type qualifiers to C,
http://www.cs.berkeley.edu/~jfoster/cqual/

[18] “Web-Based Cqual”, http://lagaffe.cs.berkeley.edu/

[19] Xiaolan Zhang, Antony Edwards, Trent Jaeger, “Using CQUAL for Static Analysis of Authorization Hook Placement”, 11th USENIX, 2002,
http://www.usenix.org/publications/library/proceedings/sec02/zhang.html

[20] Prelude,　The Open Source Quality Project(OSQ),
http://www.cs.berkeley.edu/~weimer/osq/

[21] Jeffrey S.Foster, Manuel Fähndrich, Alexander Aiken, “A Theory of Type Qualifiers”, PLDI’99, 1999,
http://www.cs.berkeley.edu/~jfoster/papers/pldi99.pdf

[22] Libparanoia Home Page, Jeffrey S.Foster, Alex Aiken, “Checking Programmer-Specified Non-Aliasing”, University of California, Berkeley, 2001
http://www.cs.berkeley.edu/~jfoster/papers/tr01-restrict.ps

[23] Jeffrey S.Foster, Tachio Terauchi, Alex Aiken, “Flow-Sensitive Type Qualifiers”, PLDI 2002, 2001,
http://www.cs.berkeley.edu/~jfoster/papers/pldi02.pdf

付録 A. ソースコード検査ツールの URL リスト

1. **RATS (Rough Auditing Tool for Security)**

   Homepage　http://www.securesoftware.com/

   ソフトウェア　http://www.securesoftware.com/auditing_tools_download.htm

2. **ITS4 (It's the Software Stupid Source Scanner)**

   Homepage　http://www.cigital.com/its4/

   ソフトウェア　http://www.cigital.com/its4/download.php

3. **Flawfinder**

   Homepage　http://www.dwheeler.com/flawfinder/

   ソフトウェア　http://www.dwheeler.com/flawfinder/flawfinder-1.21-1.noarch.rpm

4. **Splint (LCLint)**

   Homepage　http://www.splint.org/

   ソフトウェア　http://www.splint.org/download.html

5. **Cqual**

   Homepage　http://www.cs.berkeley.edu/~jfoster/cqual/

   ソフトウェア　http://www.cs.berkeley.edu/~jfoster/cqual/#download

付録 B. その他の研究プロジェクト・商用ツール

研究プロジェクト

本報告書で取り上げなかった研究プロジェクト（研究成果としてのツール）としては、以下のようなものが存在する。

1. Sardonix

Sardonix はオープンソース・ソフトウェアのセキュリティ発展の為の団体である。米国国家機関である DARPA の CHATS プログラムから支援を受けている。Sardonix では、著名なオープンソース・ソフトウェアに対するソースコード監査の結果とパッチの提供が行われている。現在の所、17 のオープンソース・ソフトウェアに対しての監査結果が公開されており、今後の監査対象として 14 のオープンソース・ソフトウェアが取り上げられている。

https://sardonix.org/

2. Open Source Quality Project

ソフトウェア品質の保証のための技術とツールの研究を行うプロジェクトである。特にオープンソースソフトウェアの品質向上のためのツールの設計・構築に焦点をあてている。様々なツールの開発が行われており、Cqual はこの研究プロジェクトの成果物のうちの一つである。

http://www.cs.berkeley.edu/~weimer/osq

3. Pscan

printf スタイルの関数に対して検査を行うツールである。printf スタイルの関数の検査に特化しており、特にフォーマット・ストリング・バグの検出に適している。

http://www.striker.ottawa.on.ca/~aland/pscan

商用ツール

本報告書で取り上げたツールの他に、商用では以下のようなツールが存在する。

1. **PARASOFT: CodeWizard**

   CodeWizard は 320 個に及ぶコーディングルールを使用して、ソースコードを静的に検証をするツールである。ルールとソースコードの状態を照合するツールである。

   http://www.techmatrix.co.jp/asq/codewizard

2. **東洋テクニカ: QAC**

   QAC は 1000 以上に及ぶプログラム上の問題チェックを行うことができるツールである。コードメトリックスの測定機能も有しており、コードの状態を定量的に測定することが可能である。

   http://www.toyo.co.jp/ss

3. **Reasoning: Illuma**

   NULL ポインターへの不正参照や変数配列外へのアクセスといった脆弱性の要因となる問題の検査を行うツールである。

   http://www.reasoning.com/

4. **Gimpel Software: PC-lint/FlexLint**

   PC-lint/FlexLint は構文解析の手法によって静的なソースコードの検査を行うツールである。

   http://www.gimpel.com/html/lintinfo.htm