平成 20 年度修士論文

バイナリコードから脅威度を推定する脆弱性検出ツールの実装と評価

電気通信大学　大学院情報システム学研究科

情報システム基盤学専攻

0753002　今井祥子

指導教員　多田　好克　　教授
渡辺　俊典　　教授
小宮　常康　准教授

提出日　平成 21 年 1 月 29 日

目次

図目次

表目次

第1章

背景と目的

近年インターネットの普及によってユーザが気軽に情報を公開できるようになった。しかし公開できるようになったのは情報だけではない。現在では自作のプログラムを公開するユーザが多数存在している。

そのようなプログラムを公開する手助けとなっているのが Vector[1]や窓の杜[2]といったようなソフトウェア・ライブラリサイト（図 1）である。これらのサイトは数万以上ものソフトウェアを有し、そこから目的のソフトウェアを検索・ダウンロードすることを可能にするサービスを提供している。プログラム作成者はこのサイトに自作のプログラムを登録するだけで簡単に公開することが出来るのに加え、ユーザもこのサイトで検索をかけるだけで多数のソフトウェアの中から目的に合ったソフトウェアを見つけることが出来る。

このようなサイトにより、自作プログラム公開の機会の増加と共に、ユーザがプログラムをダウンロードする機会も増加している。しかし、これらのサイトでは、プログラムのダウンロードを仲介はするが、プログラム自体の安全性についてはあまり関与していない。

そのため、このようなプログラムの利用が増加するにつれ、昨今ではプログラムの脆弱性が多々発見されている。ここでいう脆弱性とはプログラムのバグや欠陥といったものを指している。この脆弱性を利用することで、攻撃者がそのプログラムをフリーズさせたり、挙句にはそのプログラムが動作しているシステムを乗っ取ったりするということが可能になってしまう。コンピュータ緊

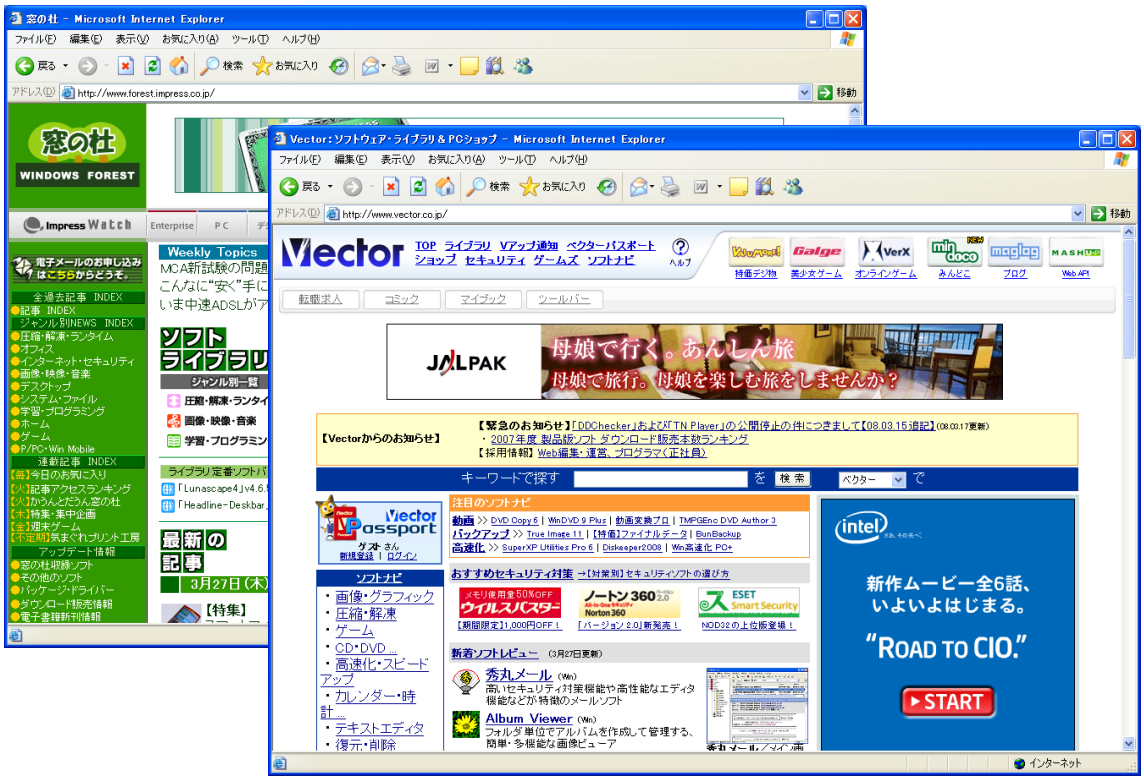

図 1：ソフトウェア・ライブラリサイト

急対応チーム／調整センター、CERT/CC[3]や IPA（独立行政法人情報処理推進機構）[4]等のサイトには多数の脆弱性が報告されており、また、広く使用されているプログラム内にも脆弱性が発見されている。

例えば解凍ツール Lhaca[5]のバージョン 1.2.0 では、バッファオーバフローの可能性がある `strcpy` 関数を使用していたため、それを利用した攻撃者がシステムを乗っ取ることが可能になってしまった。他にも iTunes や Flash Player など様々なプログラムから脆弱性は発見されている。

したがって使用前にプログラムの安全性を判断することが求められるが、それは容易ではない。なぜなら公開されているプログラムの多くが、ソースコードが公開されていないバイナリコード形式で配布されているためである。バイナリコードは、そのままではただの 0 と 1 の並びであり、どのような関数が呼ばれているかなどの情報を得ることが出来ず、安全性を判断することが困難である。

本研究では、利用前にプログラムの安全性を把握可能にするため、バイナリコードを解析してその安全性を容易に計るシステムを提案する。コード内に存在する脆弱性の原因となる関数とその使用法を分析し、そのプログラムの脅威度を点数化して分かりやすい形で出力する。

本論文は以下のような構成になっている。まず第１章で本研究に関する背景と目的を述べた。第２章では検査対象や検出可能な脆弱性、検出手法など、本システムの設計を述べる。第３章では本システムがどのような環境でどのように実装されたかを述べ、第４章では作成したシステムの使用例を簡単に説明する。第５章では本システムに対して行った実験とその考察を述べ、実用に耐えうることを示す。第６章では関連研究を紹介する。最後に第７章で本研究についてまとめる。

第2章

システムの設計

本システムは、バイナリコードを入力することでその内部に含まれた脆弱性を検出し、それにしたがって脅威度を推定、出力するものである。その概要を図 2 に示す。

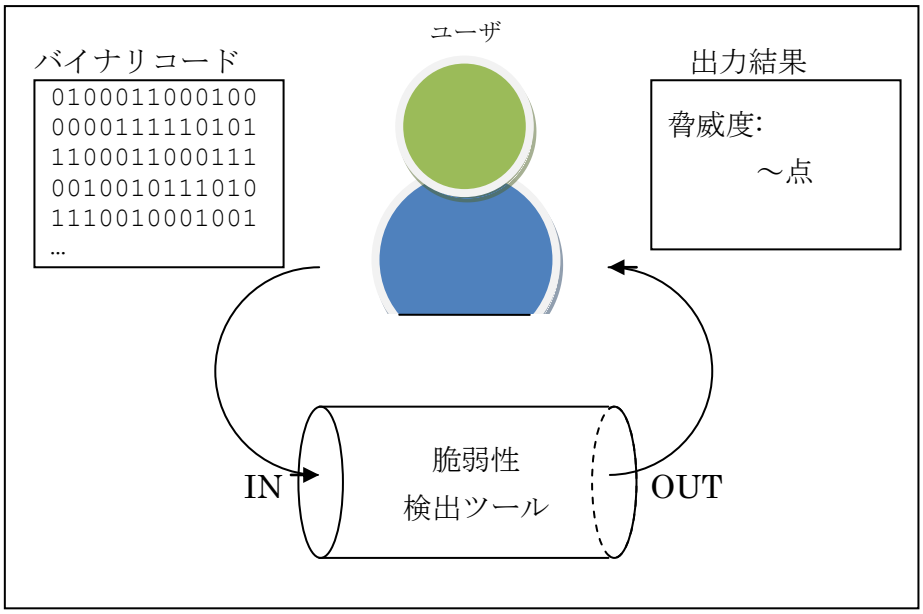

図 2：システムの概要図

以下で、本システムの検査対象についてと、脆弱性の検出方法、検出可能な脆弱性、バイナリコードの解析方法、検出結果の表示方法について述べる。

2.1 検査対象

本システムでの検査対象は、ソースコードが公開されておらずバイナリコード形式で配布されているプログラムである。バイナリコードは CPU アーキテクチャやコンパイラ、OS などの環境によって作成されるものが異なる。たとえば、バイナリコードの種類（ELF, PEI 等）や使用される CPU 命令、引数をスタックにプッシュする順番などは環境ごとに違っている。今回は x86 アーキテクチャ上の Linux にて gcc でコンパイルされたプログラムを検査対象とした。

2.2 脆弱性の検出方法

C のソースコード内に含まれる脆弱性の検出方法には、パターンマッチングによる検出と、構文解析による検出が考えられる[6]。パターンマッチングは、脆弱性を含む関数や変数の情報をデータベースとして保持し、それにしたがって検出を行う。他方、構文解析では、プログラムを構文木に変換し、それにしたがってデータ型の不一致などの検出を行う。しかし、構文解析による検出を行うと、処理が重くなってしまうとともに、ソースコードを理解したうえで様々な注釈、たとえばこのデータは null になる可能性があるなどのコメントをつけなければ正確な脆弱性検査ができない場合が存在する。

本システムの検査対象がバイナリコードであることをふまえると、内容の理解が非常に困難であることに加え、ソースコードのように注釈を付けることができないため構文解析は不向きであるといえる。また本研究では、開発者ではなく使用者が脆弱性を容易に把握することを目的としているため、内容を理解して注釈をつけなければならない構文解析は本システムの目的に反する。

加えて構文木の作成には様々なアーキテクチャ命令についての情報を保持していなければならない。今回は上述したように x86 アーキテクチャを対象としているが、将来的には移植を考えているため、できる限りアーキテクチャに依存する部分を削りたい。

したがって本研究では、構文解析ではなくパターンマッチングによって脆弱性を検出する方法を採用した。検出方法と脆弱性データベースは既存のソースコード検査ツール、ITS4[7][8]の情報を参考にしている。ITS4 を採用した理由は、オープンソースであり、かつ、C 言語で書かれたプログラムだったので、プログラム内容の把握が比較的容易であり、本研究の参考とすることが可能であったためである。

2.3 検出可能な脆弱性

パターンマッチングによる検出においては、検出可能な脆弱性は脆弱性情報を保持するデータベースの内容に依存する。本システムで参考にしている ITS4 のデータベースには脆弱性をもつライブラリ関数についての情報が保持されており、その内容は以下のように分類できる。

- バッファオーバフローを引き起こす可能性がある関数

  Ex. `gets` や `strcpy` など

  - 書き込みを行うバイト数を指定しないなどの問題で、確保したバッファの領域を越えて書き込みを行ってしまう可能性がある。重要なデータがバッファの後ろの領域に確保されている場合、そのデータが上書きされ、悪用されてしまう恐れがある。

- レースコンディションを引き起こす可能性がある関数

  Ex. `access` や `fopen` など

  - ファイルの入出力を行う時に、ファイルのチェックやファイルの処理にこれらの関数を使用すると、その処理と処理との間にわずかな隙間ができてしまう場合がある。そのため攻撃者はこのわずかな時間に処理を割り込ませ、ファイルの改変を行うことが可能になる。この脆弱性を利用すると、本来書き込み権限のないファイルに書き込みが可能になったり、閲覧が可能になったりするなどの危険がある。

- 乱数生成に関する関数

  Ex. `random` や `rand` など

  - 乱数生成に関する関数だが、これらの関数が生成するのは乱数ではなく疑似乱数であるため、値の割り出しが不可能ではない。ゆえに重要なデータ（パスワード他）としてこれらの関数により生成された乱数を使用するのはセキュリティ上避けるべきである。

- ユーザが悪意あるコードを埋め込む可能性がある関数

  Ex. `recvfrom` や `recv` など

  - これらの関数はユーザからの入力を受け付ける関数である。したがってユーザが任意に文字列を指定することができる。このような関数から取得した文字列を、たとえばデータベースにおくるクエリなどに使用した場合、ユーザが悪意を持って指定した文字列によって想定しない動作をしてしまう可能性がある。

- フォーマット文字列の脆弱性を持つ関数

  Ex. `printf` や `fprintf` など

  これらの関数はフォーマットを指定して文字列を出力する関数である。フォーマットを指定する文字列は静的に定義することが推奨されるが、ポインタを使用する場合も存在する。このとき、このポインタが指す文字列がユーザの入力によるものだった場合、スタック上にある重要なデータを見たり、上書きしたりすることが可能になってしまうため、危険である。

本システムでは、ITS4 が対象にしている上記の脆弱性が検出可能である。

2.4 バイナリコードの解析方法

バイナリコードはただの 0 と 1 の並びであるため、そのままでは解析が困難である。ゆえに、解析を容易にするために何らかの変換が必要となってくる。

本研究では先に述べたようにパターンマッチングによる検出方法を採用し、

脆弱性を含む関数の使用とその使用法を検出している。そのため、関数呼び出し命令ほか複数の命令の使用を把握する必要がある。したがってどのような命令がどのような順番で使用されるかということが容易に把握可能な逆アセンブルを行い、バイナリコードを変換することとした。

しかし、各命令における 0 と 1 のパターンをすべて記憶し変換するプログラムを書くのは容易ではない。よって、本研究では既存のツールを使用し、バイナリコードを変換することにした。

本システムの処理の流れを図 3 に示す。本研究での脆弱性検出方法は先の 2.2 節で述べたように ITS4 を参考にしている。ITS4 では、単純なパターンマッチングで脆弱性を持つ関数の使用箇所を検出するだけでなく、使用時に引数としてどのようなものが指定されたかを検証することで、その脆弱性の危険度をより正確に把握することが可能になっている。本研究ではこの引数検証方法をバイナリコードに応用している。

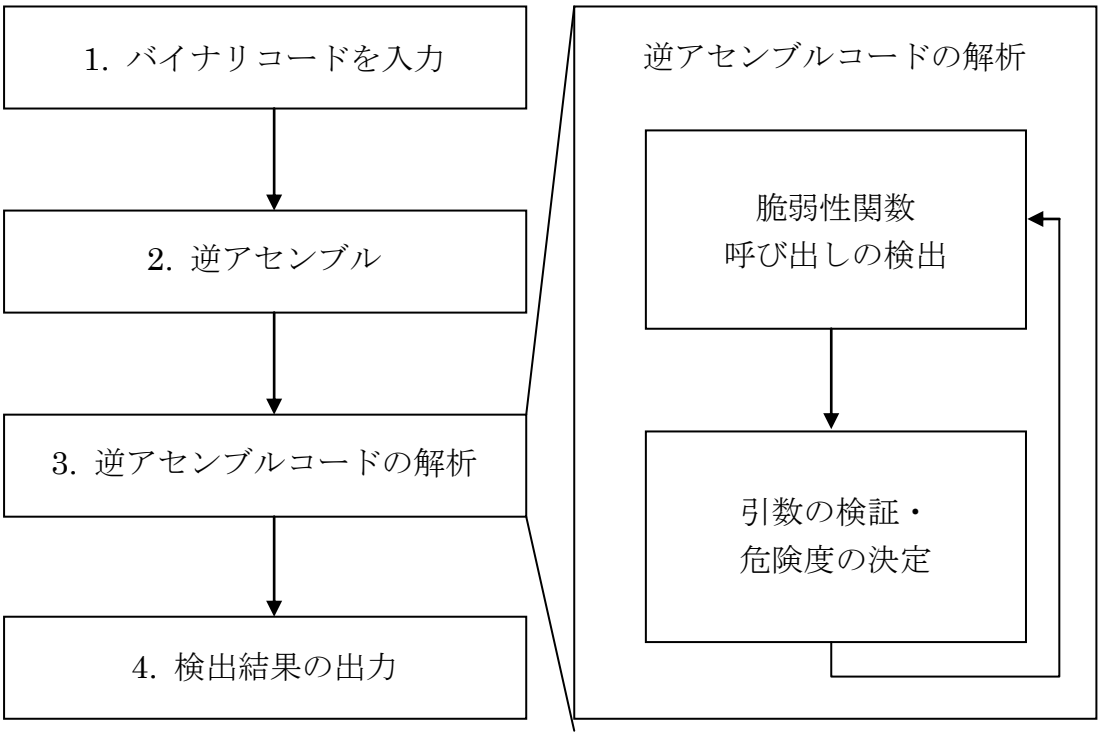

図 3：本システムの処理の流れ図

2.5 検出結果の表示方法

データベース内にある、各脆弱性関数の危険度別に個数を表示する。加えて、それぞれに重みをつけ、全体的な点数も算出する。

また、詳細な情報を知りたいといった場合に備え、オプションといった形で詳細情報を出力可能とした。この情報は主に発見された脆弱性関数の危険性についてであり、また、脆弱性関数がどこから呼ばれているのかといった情報も含んでいるため、開発者へのフィードバックとして使用可能である。

第3章

システムの実装

本システムは 2.4 節に示した通り、次のような処理の流れになっている。

1. バイナリコードの入力
2. 逆アセンブル
3. 逆アセンブルコードの解析
4. 結果の出力

以下では、本システムの実装環境と脆弱性データベースについて述べたのち、上記の処理の流れにしたがい、実装の詳細、実装上の問題点を議論する。

3.1 実装環境

本システムは、x86 アーキテクチャ上の Linux にて Java 言語を用い、実装された。プログラムの行数はおよそ 1500～1800 行程度となった。

Linux で使用されているバイナリコードのフォーマットは ELF（Executable and Linking Format）である。そのため、本システムでは ELF を対象にして検出を行っている。また、どのようなレジスタを使用しているかなどのバイナリコードの仕様は、Linux ABI（Application Binary Interface）にしたがって

いる。本論文に関係する部分としては、逆アセンブルコードを解析する際に使用するスタックポインタを保持しているレジスタが%esp であること、引数をスタック上に置く順番が後ろからであること、つまり、`foo(a, b, c);` ならば、`c, b, a` の順でスタックに積まれる（図 4）ということがあげられる。

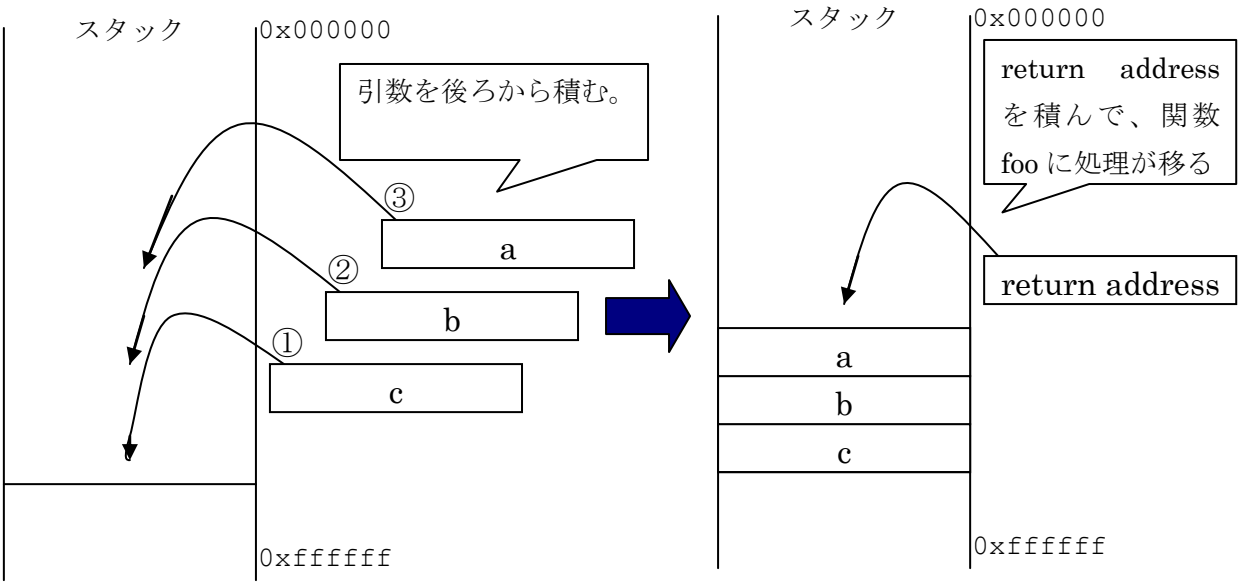

図 4：関数呼び出し時のスタック

3.2 脆弱性データベース

本システムでは 2.2 節で述べたように、パターンマッチングで脆弱性を検出するため、脆弱性に関するデータベースが必要となる。本システムで参考としている ITS4 ではそのソースコード内で構造体としてデータを保持しているため、本システムの開発言語である Java では扱いにくい。したがって、扱いやすい形式に変換することにした。

Java には XML の使用をサポートしている DOM（Document Object Model）が存在する。これを使用すると、XML の構造を把握したり、特定要素にアクセスしたりするなどの処理が容易に行える。ゆえに、本研究ではデータベースを XML 形式で保持し、DOM を使用してその読み込みを行うことにした。

そこで、ITS4 のデータベースを XML に変換した。変換は単純で以下のような ITS4 のデータ（図 5）を XML（図 6）に書き直すだけである。この変換処理を行う Java プログラムを作成し、自動的に変換を行った。

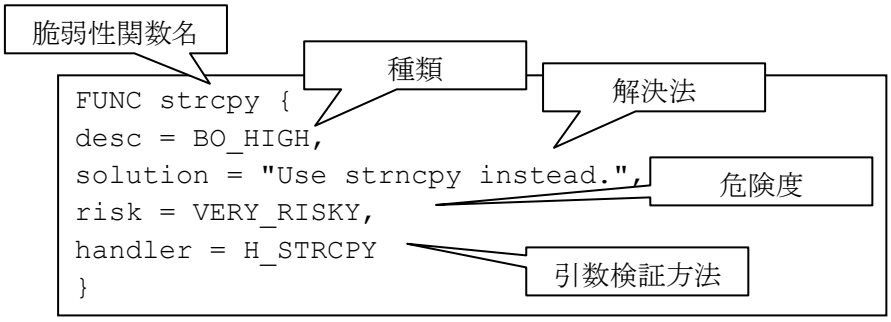

図 5：ITS4 の脆弱性 関数データの一例

```
<func desc=" BO_HIGH ">
<name>strcpy</name>
<solution>"Use strncpy instead."</solution>
<risk>VERY_RISKY</risk>
<handler>H_STRCPY</handler>
</func>
```

図 6：本システムの脆弱性関数データの一例

3.3 バイナリコードの入力

バイナリコードの入力は、実行時に検査対象のファイルを指定することにより行われる。指定するファイルは Linux における実行ファイルフォーマット、ELF 形式のファイルである。

3.4 逆アセンブル

本システムの実装環境である Linux には、ELF フォーマットのバイナリに対する既存の逆アセンブルツール、objdump が存在するため、それを利用した。objdump では、逆アセンブルしたコードの出力形式や変換箇所を、オプションとして指定することができる。3.5 節で説明するが、本研究ではバイナリコード内のテキストセクションとリードオンリーデータセクションの解析を行うため、それぞれのセクションを独立して逆アセンブルした。

具体的なオプションと出力されるコード例を以下に示す（図 7、図 8）。

- テキストセクションの変換

```
> objdump -S -j .text foo
```

```
foo:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:

0804832c <_start>:
 804832c:     31 ed                   xor    %ebp,%ebp
 804832e:     5e                      pop    %esi
 804832f:     89 e1                   mov    %esp,%ecx
 8048331:     83 e4 f0                and    $0xfffffff0,%esp
...
```

図 7：テキストセクションの逆アセンブルコード

● リードオンリーデータセクションの変換

```
> objdump -S -j .rodata foo
```

```
aa_spiral:     ファイル形式 elf32-i386

セクション .rodata の逆アセンブル:

08048c44 <_IO_stdin_used>:
 8048c44:         01 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
         ...
 8048c60:         50 6c 65 61 73 65 20 69 6e 70 75 74 20 77 69 64    Please input wid
 8048c70:         74 68 3a 20 00 25 73 00 50 6c 65 61 73 65 20 69    th: .%s.Please i
 8048c80:         6e 70 75 74 20 68 65 69 67 68 74 3a 20 00 50 6c    nput height: .Pl
 8048c90:         65 61 73 65 20 69 6e 70 75 74 20 73 70 61 63 65    ease input space
 8048ca0:         3a 20 00 50 6c 65 61 73 65 20 69 6e 70 75 74 20    : .Please input
 8048cb0:         63 68 61 72 61 63 74 65 72 3a 20 00 0a 00 00 00    character: .....
 8048cc0:         57 69 64 74 68 20 25 64 2c 20 48 65 69 67 68 74    Width %d, Height
 8048cd0:         20 25 64 2c 20 53 70 61 63 65 20 25 64 2c 20 43     %d, Space %d, C
 8048ce0:         68 61 72 61 63 74 65 72 20 25 63 0a 0a 00          haracter %c...
```

図 8：リードオンリーデータセクションの逆アセンブルコード

テキストセクションは、プログラムのコードを保持するセクションであり、この逆アセンブルを行うことで各命令の情報を容易に取得可能になった。また、リードオンリーデータセクションは、コード内で静的に定義された文字列（以下、静的文字列とする）のデータを保持しておくセクションであり、この逆アセンブルを行うことで文字列の内容を容易に把握可能になった。以上の情報は逆アセンブルコードの解析時に使用される。

3.5 逆アセンブルコードの解析

ここでは、脆弱性を含む関数を使用している箇所を検出し、その使用法を検証する。具体的には、呼び出し時に使用された引数の検証を行うことで、その危険度を調整する。

解析を行うセクションは二つであり、まず、引数の検証を行う前準備としてリードオンリーデータセクションを解析し、次に脆弱性関数の呼び出し箇所を検出するためにテキストセクションを解析する。以下では、解析の中心となるテキストセクションについて説明したのち、次にリードオンリーデータセクションについて述べる。

3.5.1テキストセクションの解析

テキストセクション内を逆アセンブルすると、プログラムのコードを知ることができる。このコードを解析して、脆弱性関数を使用している箇所を検出し、その引数を検証する。処理はテキストセクション内で定義されている各関数単位で行い、その具体的な処理の流れを図 9 に示して、それにしたがって処理を説明する。

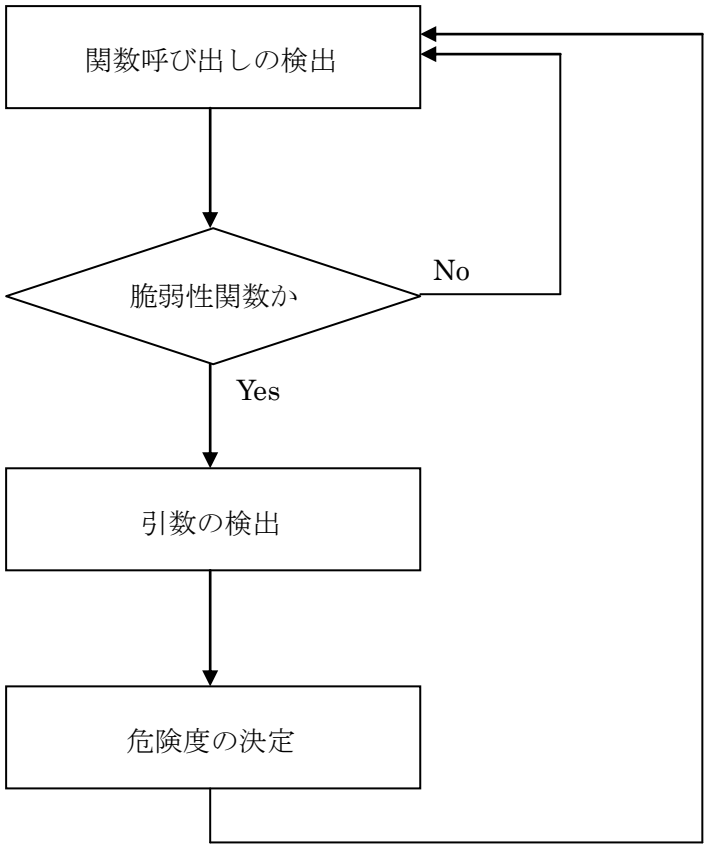

図 9：脆弱性関数検出の流れ図

3.5.1.1 関数呼び出しの検出

まず、関数呼び出しを検出する。本システムの実装環境である x86 命令では、関数呼び出しは call 命令なので、call 命令を使用している箇所を検出する。また、call 命令だけでなく jmp 命令が使用される場合もあるため、それも検出する。

3.5.1.2 脆弱性関数かどうかの判定

次に、脆弱性関数かどうかの判定を行う。関数呼び出しにて呼び出されている関数名が脆弱性データベース内に存在するかどうかを検証し、存在しない場合は再び関数呼び出しの検出に戻る。存在する場合は、次の引数の検出に移る。

3.5.1.3 引数の検出

ここでは、バイナリコードとソースコードの検査の最大の違いである、引数の検出について述べる。ソースコードでは引数は関数名のすぐ後ろに記述されているが、バイナリコードでは関数呼び出しと引数の指定が複数の命令にわたって行われる。そのため、ソースコードのように容易に検出することはできない。

まず、以下に逆アセンブルコードの例を挙げる（図 10)。この例を見ると、脆弱性関数 `printf` が `call` 命令で呼び出されているのがわかる。関数呼び出し時に指定される引数は、図 4 にあるようにスタックにおかれる。この関数呼び出し命令の前に行われているスタックへのデータ移動命令は、この例では `push` であり、`call` 命令の前に四回行われている。それぞれ移動している値とソースコードとを確認すると、3.1 節で述べたように、引数が後ろから積まれているのが確認できた。したがって、`call` 命令に最も近い `push` によってスタックへ移動された値が第一引数であり、次が第二引数といったように判別できる。

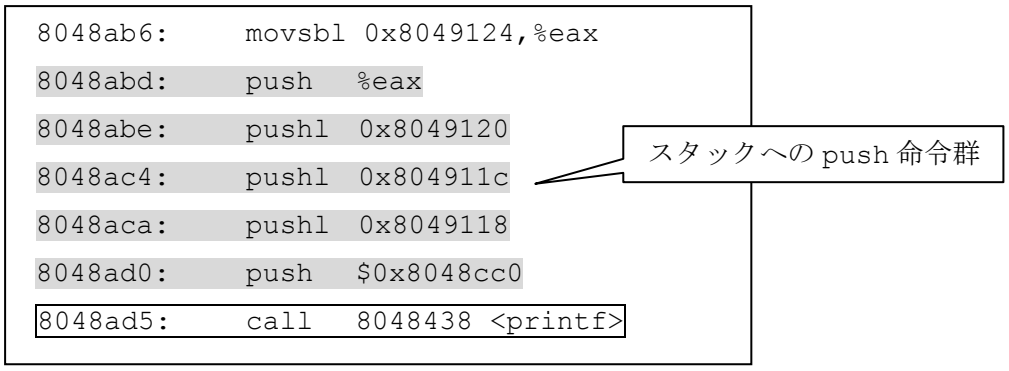

図 10：push 命令を使用して引数がスタック上に
置かれる場合のコード例

また、以下の例（図 11）では、push ではなく%esp レジスタからの相対アドレスが指す位置に、mov を使用してデータを移動していることが見て取れる。本システムの環境では、%esp はスタックポインタを表しているため、これはスタックへのデータ移動、つまり、引数を指定している命令だということが分かる。ゆえに、先ほどと同様、関数呼び出し直前にスタックへ移動されるのが第一引数だということが分かる

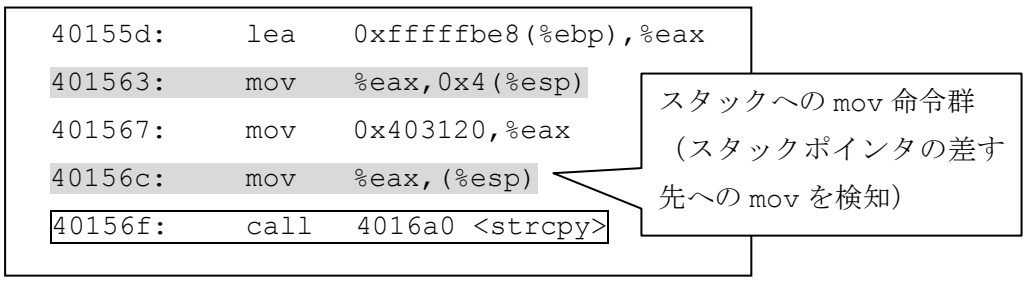

図 11：mov 命令を使用して引数がスタック上に
置かれる場合のコード例

したがって、上記二つの例から、引数の検出は call 命令のような関数呼び出し命令とその直前にある関数呼び出し命令との間にあるスタックへのデータ移動命令、push と mov を検出することとした。

また、以下の例（図 12）では、変数を使用して、静的文字列を strcpy の第二引数に指定しているソースコードを示す。この例では、strcpy に直接指定されているのは message という変数だが、実際にはその前に message に静的文字列を代入しているため、静的文字列を引数として指定しているといえる。次節で説明する引数検証方法の中には静的文字列の場合に危険度を下げるというものがあるため、この message が静的文字列だということを判別したい。

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  char buf[100];
  char *message = "This is a test message.";

  strcpy(buf, message);

  return 0;
}
```

図 12：変数を使用して静的文字列を引数に指定する場合のソースコード例

次に、図 12 のソースコードを逆アセンブルしたものを図 13 に示す。これを見ると、第二引数として引数 0xffffff84(%ebp)をスタック上に置いている。また、その前に、mov 命令を使用してデータ$0x804848c をその領域にコピーしているのがわかる。このアドレス 0x804848c が指す先には文字列"This is a test message."が入っているため、この mov 命令を検知すれば、静的文字列だということが判別可能になる。したがって、mov 命令を追跡していけば、その領域にどんなデータが入っているのかを簡単に把握することができると考えられる。

```
8048383:     movl   $0x804848c,0xffffff84(%ebp)
804838a:     sub    $0x8,%esp
804838d:     pushl  0xffffff84(%ebp)
8048390:     lea    0xffffff88(%ebp),%eax
8048393:     push   %eax
8048394:     call   80482b0 <strcpy@plt>
```

図 13：変数を使用して静的文字列を引数に指定する場合の逆アセンブルコード例

したがって本システムでは、mov 命令を使用したデータの追跡を行い、実際はどんな値が引数として指定されているのか、ある程度把握することが可能になった。

3.5.1.4 引数の検証・危険度の決定

最後に検出した引数の検証を行い、この脆弱性の危険度を決定する。引数の検証方法は先に示した通り ITS4 の手法を応用している。ITS4 での引数検証の方法は大きく分けて二つある。一つ目は静的に定義された文字列を使用しているかどうか、二つ目は同じファイルに対して操作を行っているかというものである。

まず、静的に定義された文字列を使用しているかどうかの検証について説明する。この検証を行う関数の例として strcpy を挙げる。strcpy 関数は以下のような仕様を持つ。

```
char *strcpy(char *dest, const char *src);
```

この関数は第二引数に指定した文字列を第一引数に指定したバッファにコピーする関数である。しかしコピー文字数に制限がないため、第二引数で指定された文字列の長さが、第一引数で指定したバッファの長さを超えていた場合、コピー先として確保されたバッファを越えて、その後ろにある領域に対してもデータの書き込みを行ってしまう。そのため、後ろに重要なデータを確保していた場合そのデータを損傷してしまう恐れがある。つまり、バッファオーバフローの危険性がある。ゆえに、現在は文字数制限のある strncpy 関数を使用するよう推奨されている。

この関数を使用する場合、第一引数の種類によってバッファオーバフローを引き起こす可能性が上下する。具体的にいえば、第一引数としてポインタを指定した場合は危険度が上がるが、静的文字列を指定した場合は危険度が下がる。ポインタを指定した場合、ユーザからの入力など、文字数が把握不可能な場合が存在し、第一引数として指定したバッファ内にその文字列が収まるか否かの判断が困難である。よって危険度が上がる。静的文字列を指定した場合には、その文字数を開発者が把握することが可能であり、バッファに収まるか否かの判断が比較的容易である。ゆえに危険度が下がる。これらより、静的文字列を使用しているか否かを判別することで、その脆弱性の危険度をより正確な形で判断することが可能だということが分かる。

静的文字列か否かを検証するために、本システムでは、引数として指定された値に対し、リードオンリーデータセクション内のデータとの照合を行う。リードオンリーデータセクションにはコード内で静的に定義された文字列が保持されているため、このセクション内を指すアドレスを引数として指定している場合は、静的文字列を指定していると判断することができる。また、それだけではなく、フォーマット文字列を指定する関数の場合、静的に定義された文字

列であっても“%s”指定子があればバッファオーバフローの危険性が存在する。これは“%s”を指定した場所には、任意の文字数の文字列を挿入可能であるためである。したがって引数として指定されたアドレスによって参照されているリードオンリーデータセクション内の文字列を取得し、“%s”を含んでいるかを検証して、含んでいる場合には危険度を上げている。また、“%s”は文字数を指定して使用することができるが、その場合も注意が必要となる。文字数指定の“%<数値>s”は最小の文字数を指定するだけなので、“%s”と危険性は変わりない。しかし、“%.<数値>s”では文字数の最大値を指定しているため、開発者がバッファに収まるかを判断することが可能になる。ゆえに本システムではこの場合の危険度を“%s”を含まない文字列と同等と判断している。この仕様はITS4にはないので、本システムの方が、引数検証精度が高いといえる。

また3.5.1.3節で述べているように、本システムではデータの追跡を行っているため、変数に静的文字列を代入し、その変数を引数として使用した場合にも、静的文字列を使用していると判断することが可能である。

次に同じファイルに対して操作を行っているかという検証について説明する。この検証を行う例を以下に示す（図 14）。

```
#include <stdio.h>
#include <unistd.h>

FILE *f;
int main(int argc, char *argv[]) {
  char *fname = argv[1];
  if (!access(fname, W_OK)) {
    f = fopen(fname, "w+");
  } else {
    // Do error handling.
  }
  // Write stdin to f then exit.
}
```

図 14：レースコンディション脆弱性をもつソースコードの例

このプログラムは指定したファイルに対して書き込みが可能か確認してから、そのファイルをオープンするものである。しかし、ファイルの確認とオープンの間にわずかな処理の隙間が発生してしまう。この間にファイルのすり替えが行われてしまうと、本来書き込み可能ではないファイルが書き込み可能としてオープンされてしまう可能性がある。これは access が実 UID を使用して検査を行うのに対し、open は実効 UID が使用されてしまうためである。したがって、同一ファイルに対しこの二つの操作を行うと、レースコンディション（競合状態）を利用して処理を割り込ませることで攻撃が行われてしまう可能性があり、より危険だと判断できる。ゆえに同一ファイルに対して処理が行われているかを検証することは、より正確に脆弱性の危険度を把握することにつながる。

ITS4 では、同一ファイルに対し処理を行っているかどうかを変数名で比較している。しかし本研究での検査対象はバイナリコードなので変数名は取得できない。したがって同一領域を引数にしているかどうかを比較することにした。下記に図 14 の例を逆アセンブルしたコードを示す（図 15）。

```
80483de:   push   $0x2
80483e0:   pushl  0xfffffffc(%ebp)
80483e3:   call   80482e0 <access@plt>
80483e8:   add    $0x10,%esp
80483eb:   test   %eax,%eax
80483ed:   jne    8048407 <main+0x47>
80483ef:   sub    $0x8,%esp
80483f2:   push   $0x80484e0
80483f7:   pushl  0xfffffffc(%ebp)
80483fa:   call   8048300 <fopen@plt>
```

図 15：レースコンディション脆弱性をもつソースコードを逆アセンブルした例

これにより、access と fopen の双方は領域 0xffffffc(%ebp)を引数として指定していることがわかる。本研究ではこのように同一領域を引数として指定した場合に同一ファイルに対して処理を行っていると判断することとした。

以上二つの引数検証を行い、ITS4 で決定される危険度を参考にしたがって、その脆弱性関数の使用における危険度を決定する。

3.5.2 リードオンリーデータセクションの解析

コード内で静的に定義された文字列は、このリードオンリーデータセクション内に保持されている。引数の検証時に使用するため、このデータを各文字列に分割して保持しておく。リードオンリーデータセクションを逆アセンブルした例は図 8 に示している。

逆アセンブルコードは、左にアドレス、中央に 16 進数でのデータ、右にそれを文字に変換したものが並ぶ。解析方法は簡単で、文字列の終端記号である"¥0"、つまり 0x00 で各文字列を区切り、それぞれの文字列の開始アドレスをキーとして各文字列をハッシュテーブルに保存しておくだけである。上述した引数の検証において、引数として指定されたアドレスがリードオンリーデータセクション内の文字列をさしているかを判断するため、データの保持にはリストではなくハッシュテーブルを使用した。ハッシュテーブルを使用することにより、アドレスを与えるだけでそれに応じたリードオンリーデータセクション内の文字列を効率的に取得可能になっている。

3.6 結果の出力

出力結果として、まず危険度別に表形式で個数を表示する。また、危険度それぞれに対して 0.1～5.0 の重みをつけ、それぞれの個数との積を取り、その和を点数として出力する。また、点数によって脅威度小、脅威度中、脅威度大の三段階に分けた。

詳細な情報は、ITS4 に含まれた脆弱性データベースに従ったもので、その一覧を以下に示す。

- 脆弱性関数を使用した関数名
- 脆弱性関数の名前
- 脆弱性の種類
- 脆弱性の危険度
- 解決策

以上の情報をオプションとして出力する。

3.7 問題点

本システムの引数検出方法には問題があり、それは本システムでは解決されていない。その例を（図 16）に示す。

```
snprintf(buf, foo(x), "%s", tmp);
```

図 16：引数内で関数呼び出しを行うソースコードの例

```
80483b4:      add    $0x10,%esp
 80483b7:     pushl  0xffffff80(%ebp)
 80483ba:     push   $0x80484d0
 80483bf:     sub    $0x4,%esp
 80483c2:     pushl  0xffffff84(%ebp)
 80483c5:     call   8048374 <foo>
 80483ca:     add    $0x8,%esp
 80483cd:     push   %eax
 80483ce:     lea    0xffffff88(%ebp),%eax
 80483d1:     push   %eax
 80483d2:     call   80482b4 <snprintf@plt>
```

図 17：引数内で関数呼び出しを行う逆アセンブルコードの例

関数 snprintf は第三引数が静的文字列ならば危険度が少ない脆弱性関数である。しかし、その前の第二引数で foo という関数を使用しバッファのサイズを計算している。この場合の逆アセンブルコードを（図 17）に示す。

これを見ると、snprintf の引数をスタックに移動している途中に関数呼び出しが行われているのが分かる。この場合、本システムで検出される引数は、第一引数と第二引数のみである。なぜなら、本システムの仕様では関数呼び出しと関数呼び出しの間にあるスタックへのデータの移動を検知する方法をとっているためであり、それによって foo の呼び出しを検知した時点で引数の検出が終了してしまうからである。

これを解決する方法として考えられるのが、関数呼び出しをブロック化する方法である。各関数呼び出しとその引数の指定をグループ化することで、判別が可能になると考えられる（図 18）。

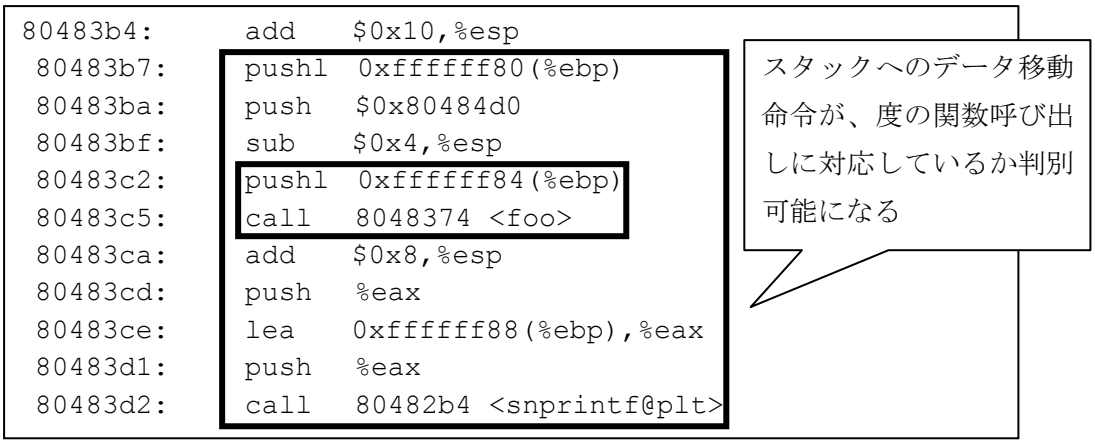

図 18：各関数呼び出しに関する命令群のグループ化

しかしこの方法は非常に困難である。まず、考えられるのは全関数の引数の個数をシステム内で保持しておく方法であるが、この方法は困難である。なぜなら、C 言語にはすでに莫大な量の関数が存在する上に、ユーザ定義された関数の引数も把握しなければならないためである。ゆえにこれを実現するのは難しい。

この問題は重要ではあるが、本システムの環境ではそれほど影響はないと考えられる。なぜなら、3.5.1.4 で説明した引数の検証に使用する引数は、第一引数から第三引数のみであるためである。検証すべき引数で関数呼び出しを使用された場合、関数からの戻り値を保持したレジスタが指定されるため、ポインタと同等と判断できる。したがって問題となるのは検証すべき引数の前で関数呼び出しを使用された場合、つまり、第一引数と第二引数で関数呼び出しが行われる場合のみである。この場合、引数を検出できなかったとして、脆弱性の危険度はポインタと同等の値に変更される。しかし、問題となるのは第一引数と第二引数のみであるということを考えると、検査結果への影響は比較的少ないと考えられる。

一方、これは引数を後ろから指定した場合であり、引数を前から指定する場合は影響が格段に上がると考えられる。なぜなら、この場合関数呼び出しを行ってはいけない引数は、第二、第三、第四引数以降の全ての引数であるため、先の場合よりも制限が厳しい。また、脆弱性関数の中にはフォーマット文字列を指定する関数も存在するため、検証すべき引数の後には多数の引数が指定される可能性がある。したがって、この環境ではこの問題は検出結果に多大な影響を及ぼすと考えられる。

しかし、本システムを実装した環境では引数は後ろから指定するため、その影響は比較的軽いと考えられる。ゆえに今回はこの問題について追及しないが、対象とするシステムによっては、さらなる引数検出方法の検討が必要である。

第4章

システムの使用例

ここではシステムの使用例を説明する。

まず、システムを起動すると以下のようなウィンドウ（図 19）が表示される。

図 19：起動時のシステムのウィンドウ

次に実行ファイルを選択する。これにはウィンドウ内の右上にある参照ボタンを使用する。参照ボタンを押すとファイル選択ウィンドウ（図 20）が開くので、そこから実行ファイルを選択し、決定する。決定後、ウィンドウ上部にあるテキストボックスに選択した実行ファイルのパスが表示される。

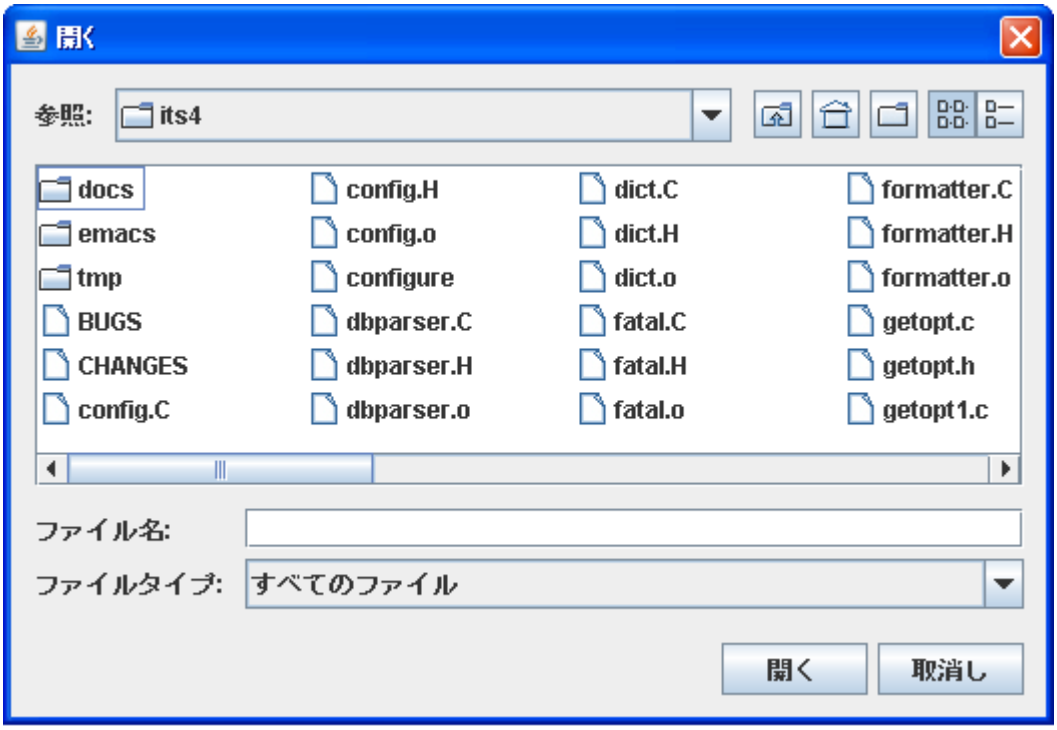

図 20：ファイル選択ウィンドウ

次に詳細情報を出力するかを選択する。詳細情報の取得を望む場合、先の実行ファイルのパスが表示されているテキストボックスの下にあるチェックボックスをチェックしておく。

選択した実行ファイルや詳細情報の有無を確認した後、解析ボタンを押すと解析が行われる。まず、ウィンドウ左中央にある表に危険度別の脆弱性関数の個数が表示される。次に、その下の点数と書かれたラベルの横に、上記の個数情報を点数化した値が表示される。そして、ウィンドウ右中央がその点数に応じて色を変える。現段階では 0～29 なら青、30～69 なら黄色、70～なら赤に変化させる（図 21）。

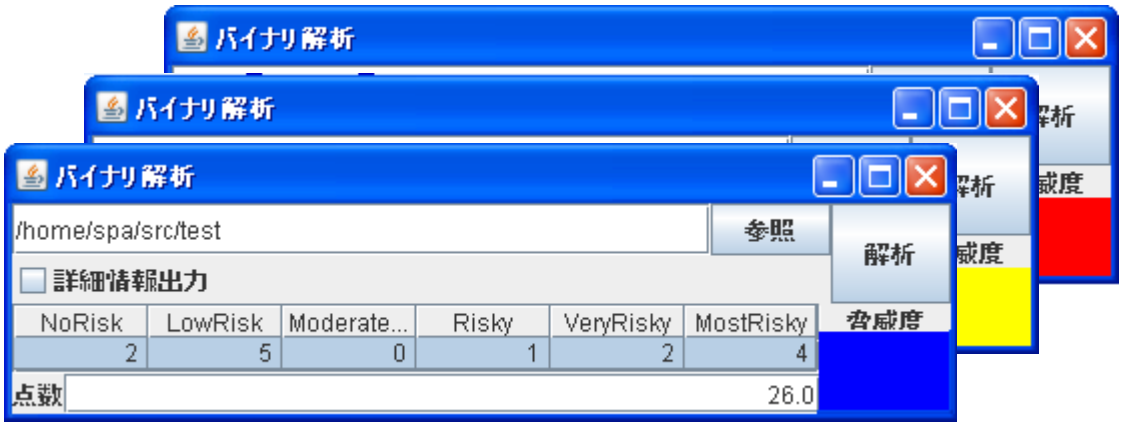

図 21：解析終了時のシステムのウィンドウ

また、詳細情報の取得を選択した場合には、新しくウィンドウを作成し、そこに詳細情報を載せている（図 22）。

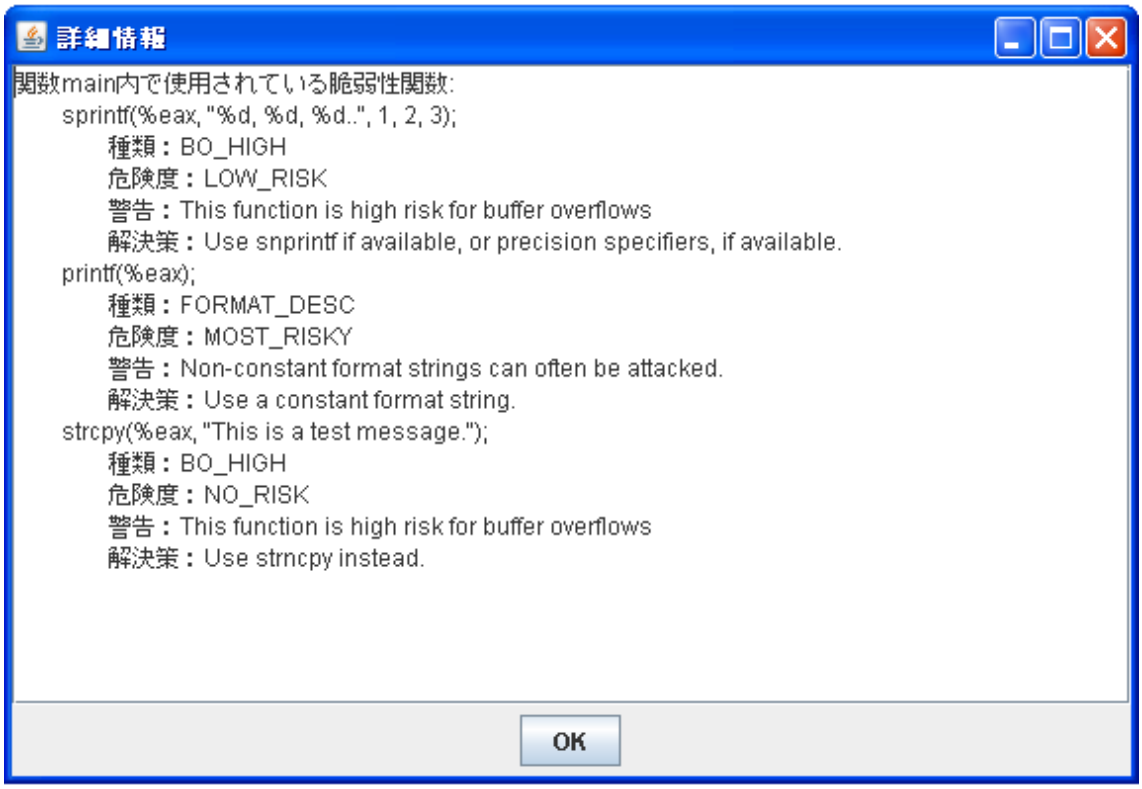

図 22：詳細情報ウィンドウ

第5章

実験と考察

本システムの評価を行うため、3 つの実験を行った。一つ目は ITS4 と本システムとの比較実験、二つ目は本システム単体の引数検出精度の検証実験、そして、実行時間の計測である。以下でそれぞれの実験における結果を示し、その考察を述べる。

5.1 実験 1　ITS4 との比較

本システムの脆弱性検出方法は、2.2 節で述べたように既存のソースコード検査ツール、ITS4 を参考にしている。そのため、ITS4 の検出精度をどの程度実現できたのかを検証する目的で、本システムと ITS4、双方のツールを使用して既存のシステムの脆弱性検出を試みた。

加えて、引数検証の制度を比較するため、複数のパターンを作成し、両ツールを使用して比較した。その結果も後に示す。

まず、既存のシステムの脆弱性検出を試み、その結果を考察した。検査対象システムは apache-1.3.9 であり、これは ITS4 の論文内で `grep` を使用した場合の検出精度比較で使用されたシステムの一つである。この apache-1.3.9 のソースコードに対して ITS4 を、それを make して作成されたバイナリコードに対して本システムを使用した。その結果を危険度別に示す（表 1、表 2）。

表 1：apache-1.3.9 を ITS4 にかけた結果

	ITS4						合計
	0	1	2	3	4	5	
apache-1.3.9	64	163	35	195	117	553	1127

表 2：make した apache-1.3.9 を本システムにかけた結果

	本システム						合計
	0	1	2	3	4	5	
ab	0	97	0	1	238	0	336
htdigest	2	15	0	1	168	0	186
htpasswd	8	26	0	2	643	0	679
httpd	0	3	0	0	3	0	6
logresolve	0	24	0	1	0	0	25
rotatelogs	0	4	0	1	1	0	6
合計	10	169	0	6	1053	0	1238

これを見ると、かなりの違いがあることが分かる。これには様々な要因が考えられる。

まず、ITS4 では危険度 5 の検出が最も多くなっているが、これは `printf` などのフォーマット文字列の脆弱性をもつ関数において静的文字列ではなくポインタを使用していると判断されたためである。しかし、ソースコードを見ると

ポインタではなく静的文字列が指定されている。ゆえに何らかのエラーによるものだと考えられる。

一方、本システムでは危険度 5 の検出量が少ないことから、きちんと静的文字列が判別され、危険度が低いということを認識できていることが分かる。

しかし、本システムでは危険度 4 の検出量が異常に多い。これはコンパイル時のオプションによってコードが複雑化したためであると考えられる。その一つに最適化オプションがある。最適化を行うと、コンパイルされたプログラムの効率化が図られるが、効率化を図るがゆえに逆アセンブルコードはより直感的に分かりにくいものに変化する。加えて、時にはソースコード内で使用した関数を別の関数に置き換えてしまう場合もある。ゆえに解析がより困難になる。その一例を以下に示す（図 23）。

```
8048ba3:     call   80489a0 <gettimeofday@plt>
8048ba8:     add    $0x10,%esp
8048bab:     movl   $0x80501a0,0xfffffffe8(%ebp)
8048bb2:     mov    0x804e460,%eax
8048bb7:     mov    %eax,0xfffffffec(%ebp)
8048bba:     cmpl   $0x0,0x804dff8
8048bc1:     je     8048beb <strcpy@plt+0x16b>
```

図 23：複雑な関数呼び出しを行う逆アセンブルコードの例

この例ではジャンプ命令を使用してライブラリ関数 strcpy へ飛んでいるのが分かる。しかし、引数をどこで指定しているのかを把握することは容易ではない。ジャンプ命令の前に add を使用してスタックポインタを移動しているため、引数は既にスタック上にあり、スタックポインタの方を引数の位置に動かすことで関数呼び出しを行っていると考えられる（図 24）。

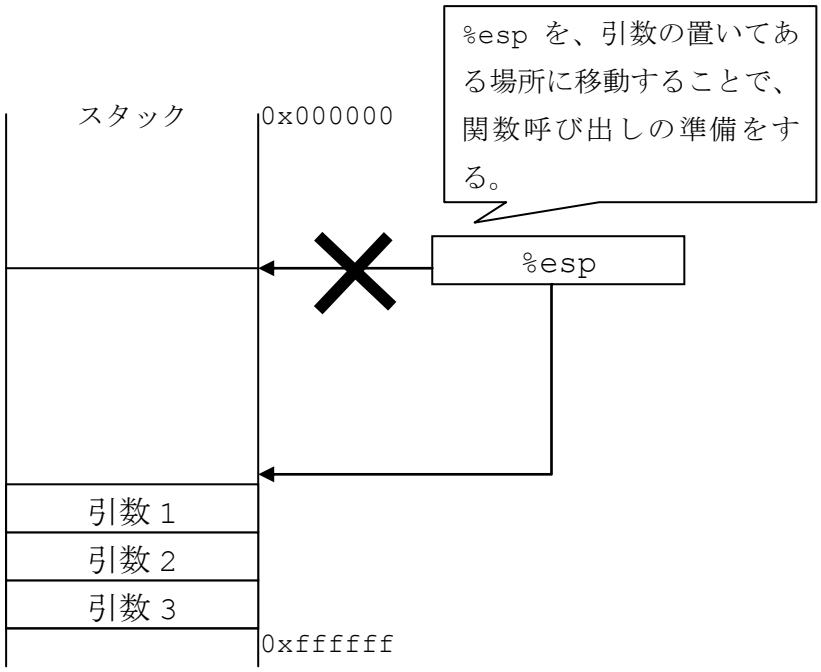

図 24：複雑な関数呼び出し時のスタック

本システムの仕様では関数呼び出しと関数呼び出しの間にある、スタックへのデータの移動を検知することで引数としているため、このような場合には引数を検知することができず、静的文字列を使用しているかどうかを検証することができない。このような場合はポインタを引数とした場合と同等と扱うこととした。

また、make して作成されたバイナリコードの一部は、ソースコード内で宣言された関数名が削除されていた。おそらく strip のようなコマンドを使用したため、関数名のようなシンボルが切り捨てられたと考えられる。ゆえに詳細情報においてどの関数から呼ばれたかを取得することが出来なかった。

次に ITS4 との引数取得精度を比較する実験を行った。使用したパターンは以下の通り。

- 引数の検証で、静的文字列かどうかを判別する関数を使用する
  - 検証する箇所の引数に静的文字列とポインタを指定
- レースコンディションの可能性がある処理を行う
- 静的文字列を変数に格納し、それを引数として指定する
- 引数内で分岐を行う
  - ex. `printf((x % 2 == 0) ? "even" : "odd");`
- 引数内で代入を行い、それを使用して静的文字列を指定する

以上に対して ITS4 と本システムを使用した。ただし、バイナリを作成する際に最適化は行わなかった。その結果を以下に示す（表 3)。

表 3：ITS4 と本システムの引数検出精度の実験結果

	ITS4	本システム
静的文字列が取得できているのか		
printf で第一引数に静的文字列を指定した場合	×	○
printf で第一引数にポインタを指定した場合	○	○
strcpy で第二引数に静的文字列を指定した場合	○	○
strcpy で第二引数にポインタを指定した場合	○	○
snprintf で第三引数に静的文字列を指定した場合	○	○
snprintf で第三引数にポインタを指定した場合	○	○
文字列の内容が把握できているか		
printf で"%s"を使用した場合	○	○
printf で"%10s"を使用した場合	×	○
printf で"%.10s"を使用した場合	×	○
レースコンディションの可能性がある処理を検知できるか		
access と fopen を使用した場合	○	○
変数に格納された静的文字列を取得できるか		
静的文字列を変数に格納し、 それを printf で使用した場合	×	○
引数内で分岐を使用して指定された静的文字列を取得できるか		
printf の第一引数を分岐にして 静的文字列を指定した場合	×	△
引数内で代入が行われた静的文字列は取得できるか		
printf で静的文字列を直接代入した場合	×	○
printf で変数経由で代入を行った場合	×	○

まず、静的文字列の取得についてだが、これは双方ともほとんど取得できている。しかし先の実験でもそうだったが、ITS4 で printf の第一引数の検証時に何かエラーが起きているのか、静的文字列を指定しても検知することはできなかった。一方本システムではどの場合でも静的文字列を取得できていることがわかる。

次に文字列内容の把握だが、ITS4 が"%s"を指定した場合以外は取得できていない。しかし、本システムはどの場合でも把握できているので精度が高いといえる。

また、レースコンディションの検知では、静的文字列の取得と同様、双方とも検知できた。

変数に格納された静的文字列を判別することについては、本システムのみ判別可能だった。これは本システムで行っているデータの追跡によるものであり、ITS4 ではデータ追跡を行っていないため静的文字列を取得できない。

また、引数内で分岐を使用して指定された静的文字列を検知できるか否かについては、ITS4 はできていないが本システムでは検知できていた。しかし、これは正確ではない。下記にテストしたソースコード（図 25）と逆アセンブルコード（図 26）を示す。

```
printf((x % 2 == 0) ? "even" : "odd");
```

図 25：分岐によって静的文字列を指定するソースコード例

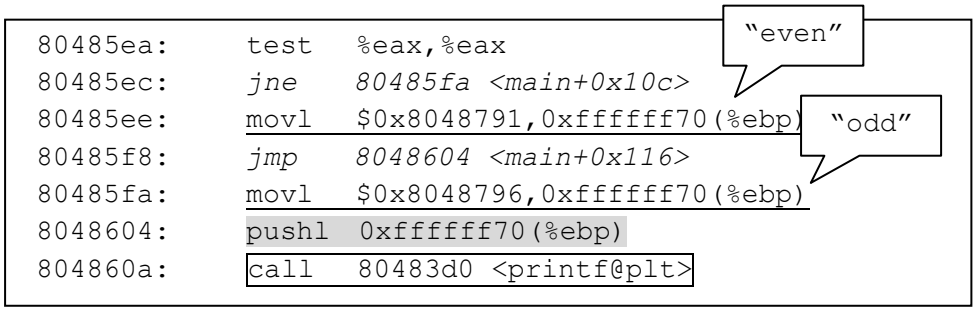

図 26：分岐によって静的文字列を指定する逆アセンブルコード例

この例では、printf の第一引数で分岐を行っている。逆アセンブルコードでは第一引数として指定されているのは 0xffffff70(%ebp)であるが、その少し前で mov を使用してその領域に静的文字列を代入している。前後にジャンプ命令があるため、どちらか一つの mov が実行されたのち、引数の push が行われる。

本システムでは mov によるデータ追跡を行っているため、push の前にある"odd"を取得することができるが、0xffffff70(%ebp)に代入されているのが静的文字列だと分かった時点で追跡が終了するため、"even"の代入は検知できない。したがって、本システムの仕様では引数指定命令により近い位置にある代入命令しか検知できない。つまり、分岐の場合、代入の順番はコンパイラや最適化オプションによって異なるものの、どちらか片方しか検出できないということになる。しかし、分岐ということはどちらも実行時には選択される可能性があり、その場合より危険な方に評価を合わせるのが妥当であると考える。ゆえに、このテストパターンでは正確な危険度を推定できているが、通常本システムでは分岐を使用した場合に正確な評価が行えないと判断する。

最後に引数内で変数に静的文字列を代入し、それを引数として指定する場合、二通りのパターンを試した。一つ目の直接静的文字列を変数に代入する場合のソースコード（図 27）と逆アセンブルコード（図 28）は以下のようになる。

```
printf(conststr = "This is a test message.");
```

図 27：引数内で代入を行うソースコード例 1

```
8048615:     movl   $0x804879a,0xffffff84(%ebp)
804861c:     push   $0x804879a
8048621:     call   80483d0 <printf@plt>
```

図 28：引数内で代入を行う逆アセンブルコード例 1

逆アセンブルコードを見ると、引数内で代入を行った場合は、代入先の変数が引数として使用されるのではなく、代入される静的文字列の方を引数として使用しているのがわかる。ゆえに、静的文字列か否かの判別は容易である。また、たとえ引数として使用されたのが変数の方であった場合でも、データ追跡によって検知が可能である。

二つ目は、静的文字列を代入した変数をさらに別の変数に代入して、それを引数として指定した場合である。そのソースコード（図 29）と逆アセンブルコード（図 30）を以下に示す。

```
char *conststr = "This is a test message.";
char *buf2;
printf((buf2 = conststr));
```

図 29：引数内で代入を行うソースコード例２

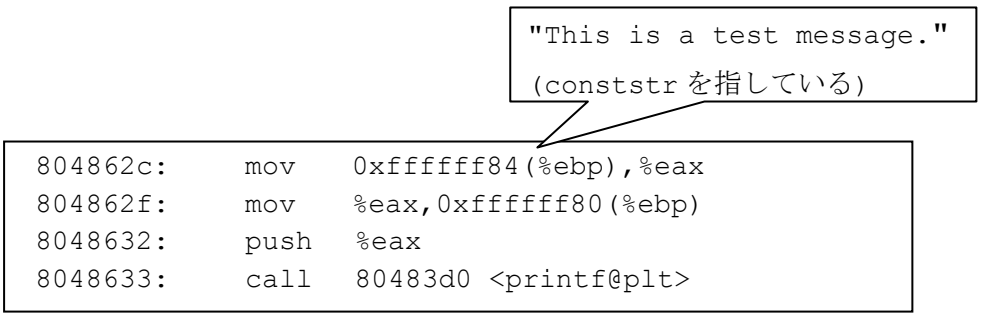

図 30：引数内で代入を行う逆アセンブルコード例２

この場合も代入先ではなく代入される値の方を引数として使用しているのがわかる。また、データ追跡によって%eax が conststr を指しているということを把握するのは容易である。ゆえに、このパターンでも静的文字列を使用していると判断できたことが分かる。

したがって、この実験結果により、引数検証方法においては、本システムはデータ追跡によってITS4よりも精度が高いといえる。

5.2 実験２　引数検出精度の検証

次に、本システム単体で行う引数検出制度の検証実験の結果を示す。この実験では先の実験1でITS4と比較する際に使用したパターンのうち、本システムで検出可能だったものを流用した。ただし、先の実験では最適化オプションを0としていたが、本実験では最適化オプションを0から4まで変化させて検証した。また、文字列内容の把握については最適化の影響を受けないため除外した。その結果が表 4である。

表 4：引数検出精度の検証実験結果

	0	1	2	3	4
静的文字列が取得できているのか					
printf で第一引数に静的文字列	○	×	×	×	×
printf で第一引数にポインタ	○	○	○	○	○
strcpy で第二引数に静的文字列	○	×	×	×	×
strcpy で第二引数にポインタ	○	○	○	○	○
snprintf で第三引数に静的文字列	○	○	○	○	○
snprintf で第三引数にポインタ	○	○	○	○	○
レースコンディションの可能性がある処理を検知できるか					
access と fopen を使用	○	○	○	○	○
変数に格納された静的文字列を検知できるか					
静的文字列を変数に格納し、それを printf で使用	○	○	○	○	○
引数内で代入が行われた静的文字列は検知できるか					
printf で静的文字列を直接代入	○	○	○	○	○
printf で静的文字列を格納した変数を使用して代入	○	○	○	○	○

まず、最適化度 0 の場合はどのパターンも検知が可能だった。

次に、最適化度 1 になると、パターンによっては検出が正しく行えなかった。まず、printf で第一引数に静的文字列を指定する場合だが、この場合は最適化によって printf がフォーマット文字列を使用しないより安全な puts に置き換えられていたため、検出できなかった。次に、第二引数が静的文字列の場合だが、この場合も最適化によって strcpy が使用されていなかった。ゆえに検出できなかった。残りのパターンは検出できたが、代入を行った場合には引数をスタック上に置くのに、これまでの関数呼び出しと異なり、push 命令ではなく mov 命令が使用されていた。

最適化度 2 の場合も最適化度 1 と変わらず、それ以上の最適化度 2〜4 も結果は変わらなかった。逆アセンブルコードも変化しておらず、これは単純なパターンを使用したためだと思われる。

したがって、実験 1 のようなジャンプ命令を使用し、引数をスタックに移動するのではなくスタックポインタを移動する関数呼び出しでなければ、本システムにはある程度の検出精度があると判断することができる。

5.3 実験 3　実行時間の測定

実験 1 で使用した apache に対して本システムを適用した時の実行時間を、バイナリファイル別に測定した。ただし、逆アセンブルツールである objdump の実行時間は差し引いた。これは本システム単体のアルゴリズムの実行時間を計測するためである。結果を以下に示す（表 5）。

表 5：実行時間測定結果

バイナリファイル	サイズ KB(Byte)	実行時間（ミリ秒）
rotatelogs	4.6 (4712)	265
logresolve	7.0 (7192)	327
htdigest	12.4 (12720)	531
ab	21.5 (22040)	639
htpasswd	32.7 (33488)	1218
httpd	458 (468964)	1982
合計	536.2 (549116)	4962

これにより、やはりバイナリコードのサイズが大きいほど処理に時間がかかっていることが分かる。しかし、rotatelogs から htpasswd まではほぼ一次元的に増加しているのに対し、httpd では htpasswd とのサイズの差から考えても増加の度合いが少ない。これは、httpd はその大きさに比べて検出された脆弱性が極めて少なく、それによって実行時間が減少したのだろうと考えられる。ゆえに、本システムの実行時間はサイズだけでなく、発見される脆弱性の量にも影響を受けていることがわかる。また、合計して 536.2KB で 4962 ミリ秒なので、

既存のシステムに対して使用しても実用に耐えうる程度の時間だと判断できる。

また、objdump の実行時間は合計 885 ミリ秒だった。

第6章

関連研究

ソースコード検査ツールとしては、まず ITS4 が挙げられる。ITS4[7][8]は先に述べたようにオープンソースのソースコード検査ツールであり、パターンマッチングで脆弱性関数を検出している。対象となる脆弱性関数はバッファオーバフローやレースコンディションを引き起こす関数、乱数生成に関する関数、ユーザが悪意あるコードを埋め込む可能性がある関数、フォーマット文字列の脆弱性をもつ関数で、とくにレースコンディションの検知に力を入れている。

また、パターンマッチングを採用したオープンソースの検査ツールとしては、flawfinder[9]というツールも存在する。このツールが対象としている脆弱性は ITS4 と同様である。Python という言語で記述されているため、やや処理が重い傾向にある。

構文解析を使用した脆弱性検査ツールには splint[10]が挙げられる。splint はソースコードを構文解析する lint というツールから派生したもので、データの遷移を解析し、それに関する脆弱性を検出する。ただし、これを使用するためにはソースコードに様々な注釈をつける必要がある。例えば、あるデータが null になる可能性があるなら、それをソースコード内に記述しておくと、そのデータを使用する前に null かどうかのチェックを行っているかを検証してくれる。

しかし、以上のツールはソースコードを検査対象としているため、ソースコードが公開されていなければプログラムの安全性を判断することが出来ない。

バイナリコードの脆弱性検出については、Tevis ら[11]が Windows における

バイナリコードの脆弱性検出を試みた。彼等は、逆アセンブルは行わずヘッダ情報のみから脆弱性を検査している。脆弱性関数の使用も検出するが、ヘッダ情報のみでは引数の検証が不可能であるため、誤検出が多く含まれてしまう可能性がある。

第7章

まとめ

インターネット上には有益なプログラムが多数公開されている。しかし、それらのプログラムには脆弱性が含まれている可能性があり、ユーザに被害が及ぶ場合も存在する。ゆえにそのプログラムの安全性を使用前に把握することが望まれるが、それは困難である。本研究ではその安全性を容易に計るシステムを提案した。それにより、プログラム内に含まれる脆弱性を検知し、点数化することでその安全性を推測することが可能になった。

本システムでは、パターンマッチングを行って脆弱性関数を使用している箇所を検出した。バイナリコードはただの 0 と 1 の並びであるため、関数の使用を容易に検出するために、既存のツールを使用して逆アセンブルを行った。これにより、解析を容易にすることが可能になった。また、逆アセンブルしたコードのうち、解析に使用したのはテキストセクションとリードオンリーデータセクションの二つである。テキストセクションにはプログラムのコードが格納されているため、脆弱性関数の使用やその使用法を検証するために解析した。リードオンリーデータセクションには静的文字列が格納されており、使用法の検証を行う前準備としてそのデータを取得するため解析した。

今後の展望としては、5.1 節で述べたように、本システムはまだ最適化等による複雑化したコードに対応できていないため、それへの対応が考えられる。また、現在はアーキテクチャ依存を避けるためにパターンマッチングで検査を行っているが、対象を今回の実装環境に限定するならばフロー解析による精密な

脆弱性解析を行うように改良することも考えられる。

また、今回は Linux を対象に実装を行なったが、将来的には他環境への移植が考えられる。x86 上の windows で使用されているバイナリの環境は、ファイルのフォーマットが PEI である以外は本システムの開発環境（引数の移動順など）と同じなので、現在のシステムを適用することが出来ると考えられる。しかし本システムで使用している逆アセンブルツールの `objdump` は Windows には存在しないツールである。また、静的解析を行うにあたり、本システムは大きく `objdump` に依存している。ゆえに、Windows 上に Linux 環境を構築する cygwin のようなシステムを利用して `objdump` を実行すれば、本システムは Windows でも問題なく動作すると考えられる。また、現在の開発環境と異なる仕様の OS に対しては、プログラムの改良が必要となる。スタックポインタの変更は容易だが、引数の検出に問題がある。現在の環境は後に指定した引数を先にスタックに移動することを前提に実装している。しかし、3.7 節で説明したように、先に指定した引数を先にスタックに移動する場合において、検証すべき引数が検出出来ない可能性が大幅に増加すると考えられる。そのため、引数の移動順が後者の場合、移植にあたってはさらなるアルゴリズムの工夫が必要になるだろう。

謝辞

本研究を行うにあたっては、さまざまな方々にお世話になりました。特に指導教員の多田好克教授や佐藤喬助教には、お忙しい中、たくさんの御指導、御助言を頂きました。ここに厚く御礼申し上げます。

また、本研究を進めることができたのは、多田研、小宮研、村山研、水野研の先生方、学生諸君のおかげでもあります。研究に関する議論への参加や多数の御助言について、心より感謝致します。

参考文献

[1]. Vecttor,
http://www.vector.co.jp/

[2]. 窓の杜,
http://www.forest.impress.co.jp/

[3]. CERT/CC,
http://www.cert.org/

[4]. IPA,
http://www.ipa.go.jp/

[5]. Lhaca,
http://park8.wakwak.com/~app/Lhaca/

[6]. 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,
http://www.ipa.go.jp/security/fy14/reports/oss_security/part2.pdf

[7]. ITS4,
http://www.cigital.com/its4/

[8]. Amitabh Srivastava and Alan Eustace, “Token-based scanning of source code for security problems,” ACM SIGPLAN Notices Volume 39, Issue 4(April 2004) Best of PLdl 1979-1999 SPECIAL ISSUE:1994 pp.528-539.

[9]. flawfinder,
http://www.dwheeler.com/flawfinder/

[10]. splint,
http://www.splint.org/

[11]. Jay-Evan J. Tevis and John A. Hamilton, Jr., “Static analysis of anomalies and security vulnerabilities in executable files,” Proceedings of the 44th annual Southeast regional conference, pp.560-565, 2006.