無害のバグを大量に含ませるプログラム難読化

大山恵弘[†] 甲斐朋希[†] 中村燎太[†]

プログラムを解析して潜在的な脆弱性を検出する脆弱性検査ツールが多数開発されている．脆弱性検査ツールは，通常，脆弱性を早期に検出して攻撃前にプログラムを修正するという良い目的に利用される．しかし，悪意ある者が，攻撃可能な脆弱性を効率的に発見する用途に悪用することもできる．ツールを用いた攻撃者による脆弱性発見を妨害する技術があれば，攻撃を成功させるコストが上がり，攻撃を減らせる可能性がある．本論文では，脆弱性検査ツールによる脆弱性発見を妨害する方式を提案する．その方式は，ソースプログラムを変換して，脆弱性検査ツールが検出するが攻撃には利用できないバグを大量に加える．加えられたバグに対して脆弱性検査ツールは大量の警告を出すため，真の脆弱性がもしあったとしても，より目立たなくなる．

Program Obfuscator for Injecting Numerous Harmless Bugs

Yoshihiro Oyama ,[†] Tomoki Kai [†] and Ryota Nakamura[†]

A number of vulnerability checkers, which analyze a program and detect potential vulnerabilities, have been developed. Vulnerability checkers are usually used for good purpose: early detection of vulnerabilities for patching programs before being exploited. However, malicious persons can also misuse the checkers to find out exploitable vulnerabilities efficiently. A technology that obstructs scanning operations by attackers will increase the cost needed for successful attacks and consequently reduce attack attempts. In this paper, we propose a scheme for obstructing the operations of vulnerability detection using vulnerability checkers. The scheme transforms a source program and injects numerous bugs that are detected by vulnerability checkers but cannot be exploited. Since a vulnerability scanner outputs plenty of warnings against the injected bugs, actual vulnerabilities, if any, become more inconspicuous.

1. はじめに

ソフトウェアの脆弱性を早期に発見するために，脆弱性を含む可能性がある箇所を自動的に検出するツール（脆弱性検査ツール）が広く利用されている[1)~5)]．ソフトウェアの開発者や配布者はソフトウェアの配布前に脆弱性検査ツールを用いて安全性を高めることができる．しかし残念ながら，脆弱性検査ツールは攻撃者が悪用することもありうる．攻撃者は，脆弱性検査ツールを用いて効率的に脆弱性を探すことにより，攻撃を成功させるコストを下げることができる．もし，脆弱性検査ツールを用いた脆弱性発見を妨害する技術があれば，そのコストを上げることができ，攻撃を減らせる可能性がある．

本研究では，攻撃者が脆弱性検査ツールを用いて脆弱性を発見する作業を妨害することにより安全性を向上させる方式を提案する．この方式では，ソースコードを変換器で変換し，一見すると脆弱性に見えるが実際は無害であるコードをプログラム中に大量に含ませる．例えば，バッファオーバーフローを起こすが攻撃者が制御を奪えないバグを含むコードを加える．なお，コードを加える際には，プログラムの動作を変えないようにする．この変換器で変換された後のソースコードに対して脆弱性検査ツールを用いた場合，変換器が加えたバグのコードが，脆弱性の可能性があるコードとして，大量に検出される．その結果，悪意ある者が脆弱性検査ツールを用いても，見せかけの脆弱性が大量に検出されてしまい，真の脆弱性が発見しにくくなる．攻撃者がそのソフトウェアを攻撃することをあきらめるか，より多くの手間と時間を攻撃に使う結果になれば，安全性を向上させる効果はあったと言える．

本研究では，C 言語のソースコードを受け取り，無害なバグを含ませた C 言語のソースコードを出力する変換器 Halebi (HArmLEss Bug Injector) を実装した．さらに，実験により，変換後のプログラムに対して既存の脆弱性検査ツールがどのような警告を出すかを調査した．

一般に，プログラムを読みにくくする変換は難読化と呼ばれる．本研究の変換器は難読化ツールの一種とみなせる．攻撃者による脆弱性発見を妨害することは，既存の難読化ツールによっても可能である．人間がプ

† 電気通信大学
The University of Electro-Communications

ログラムの動作を理解しにくくなるため，攻撃者が解析にかける時間と手間を増大させることができる．しかし，多くの難読化ツールはプログラムの意味を変えないので，脆弱性検査ツールに対して影響を与えない可能性が高い．一方，本研究の変換は，脆弱性検査ツールが出す警告を大きく増加させることができる．

長期的には様々な言語のプログラムに様々な種類のバグを含ませる方式を構築することを予定している．ただ，現時点では，バッファオーバフロー脆弱性に見せかけるバグを C 言語のプログラムに含ませる変換についてのみ研究を行った段階である．

2. 脆弱性検査ツール

プログラムの文面の単純なパターンマッチで脆弱性を検出するツールが複数存在する．RATS[3] は，C, C++, Perl, PHP, Python, Ruby などの言語のソースコードを検査できる脆弱性検査ツールである．プログラムの文面と脆弱性データベースとの間でパターンマッチを行うことにより脆弱性を検出する．ITS4[5] は，C/C++で書かれたソースコードのための脆弱性検査ツールである．検査は，ソースコード中で使用されている関数と脆弱性データベースとのパターンマッチングにより行われる．Flawfinder[2] は C/C++で書かれたソースコードの脆弱性検査を行うツールである．脆弱性を含ませる可能性が高いライブラリ関数呼び出しを発見する．これらのツールは高速であるが，プログラムの意味を考慮した解析を行わないため，精度が低いという問題がある．よって，一定以上の技術を持った攻撃者がこれらのツールを利用する可能性は低い．なお，これらのツールに大量に偽の警告を発生させるには，危険とされているライブラリ関数を無駄に大量に呼び出すコードを加えれば良い．

プログラムの意味を考慮した解析を行う検査ツールも存在する．それらは必ずしも脆弱性検出に特化したものではないが，脆弱性検出にも利用可能である．Splint[4] は C 言語のソースコードを検査し，脆弱性に限らない広範囲のバグを静的に検出するツールである．Microsoft 社の Visual Studio の上位版は「静的コード分析」と呼ばれる，バグ検出を支援する機能を備えている．gcc コンパイラも，コンパイル時にバグを検出する機能を提供している．Coverity SAVE[1] は制御フローを意識した静的解析により高い精度でバグを検出する商用ツールである．これらのツールや機能は効率的な脆弱性発見に極めて有用であるため，攻撃者が悪用することは十分考えられる．

3. 提 案 方 式

3.1 概　　要

提案方式による変換器は，ソースコードを入力として受け取り，プログラムの動作を変えないようにしつつも，大量の無害なバグを加えたソースコードを出力する．例えば，バッファの範囲外のメモリ領域にデータを書き込むが，それによってプログラムが通常と違う動作をしたり，攻撃者が制御を奪うことはないようなコードを加える．ソフトウェアの開発者または配布者は，変換後のソースコード，または，それをビルドした実行型バイナリを配布する．変換前のソースコードは配布しない．

```
int main(void)
{
  char buf1[80], buf2[40];  FILE *fp;
  ...
  fgets(buf1, 80, fp);
  /* 真のバッファオーバーフロー脆弱性 */
  strcpy(buf2, buf1);
  ...
}
```

図 1　変換前のプログラム
Fig. 1　Sample program before transformation

バッファオーバフロー脆弱性に見せかけた無害なバグを含ませる方法は複数存在するが，以下では，Halebi に既に実装されている方法を説明する．その方法では，バッファオーバフローにより上書きされる領域のデータをあらかじめ退避しておき，バッファオーバフロー後に書き戻す．変換器は大まかには以下の 3 つの処理をプログラムに追加する．

(1)　オーバーフローさせるバッファの後ろにあるデータを別の場所に退避する
(2)　バッファをオーバーフローさせる
(3)　退避していたデータを元の場所に書き戻す

2 つ目と 3 つ目の処理が，脆弱性検査ツールによってバッファオーバーフロー脆弱性と検出される可能性がある．しかし，実際はバッファの後ろにあるデータ（リターンアドレスやフレームポインタを含む）は元に戻るため，バッファオーバフロー攻撃は決して成功しない．

変換の様子を例を用いて説明する．**図 1** は，変換前のプログラムである．このプログラムには真のバッファオーバフロー脆弱性がわかりやすい形で含まれている．よって，多くの脆弱性検査ツールはこの脆弱性を検出する．

Halebi による変換により，上記の脆弱性を目立たないようにできる．変換後のプログラムを**図 2** に示す．このプログラム内には，別のバッファオーバーフローの処理が加わっている．まず，バッファオーバフローを行うためのダミーのバッファが，関数の局所変数宣言部で宣言される．ダミーのバッファは 2 つ宣言されている．`JLHvPkBWeedy3d` はスタック上バッファのオーバーフローを起こすために使われる．`NQzo1kmkHaJlU7` はヒープ上バッファのオーバーフローを起こすために

```
char yPiwiJ2fkGe[224];
char h8Lpa4Z2TyVs[240];
int main(void)
{
  char buf1[80], buf2[40];  FILE *fp;
  /* スタック上バッファ */
  char JLHvPkBWeedy3d[8];
  /* ヒープ上バッファ */
  char *NQzo1kmkHaJlU7 = (char *)malloc(8);
  /* データの退避 */
  memcpy(yPiwiJ2fkGe, JLHvPkBWeedy3d, 222);
  /* データの退避 */
  memcpy(h8Lpa4Z2TyVs, NQzo1kmkHaJlU7, 238);
  /* スタック上バッファのオーバフロー */
  strcpy(JLHvPkBWeedy3d, "XTyZE53U18Pk4");
  /* ヒープ上バッファのオーバフロー */
  strcpy(NQzo1kmkHaJlU7, "Duka5y3cuPSS");
  /* 退避したデータの書き戻し */
  memcpy(JLHvPkBWeedy3d, yPiwiJ2fkGe, 222);
  /* 退避したデータの書き戻し */
  memcpy(NQzo1kmkHaJlU7, h8Lpa4Z2TyVs, 238);
  fgets(buf1, 80, fp);
  strcpy(buf2, buf1);
  free(NQzo1kmkHaJlU7);
}
```

図 2 変換後のプログラム
Fig. 2 Sample program after transformation

使われる．関数中の 2 行の strcpy 関数の呼び出しによって，ダミーのバッファに，バッファサイズ以上のデータを書き込み，バッファオーバフローの処理を実現する．スタックとヒープの両方にバッファを確保するのは，様々な脆弱性のパターンを作るためである．バッファオーバフローの前後には，上書きされるメモリ領域のデータを退避，書き戻しする処理を出力する．退避のためのメモリ領域が，大域変数 yPiwiJ2fkGe と h8Lpa4Z2TyVs として宣言されている．最初の 2 つの memcpy 関数の呼び出しでは，それぞれ，スタック上バッファのオーバフローとヒープ上バッファのオーバフローで上書きされるデータを待避している．次の 2 つの memcpy 関数の呼び出しでは，退避したデータを書き戻している．関数末尾には free 関数によるメモリ解放の処理が追加されている．ヒープから確保されたダミーのバッファはここで解放される．

3.2 変換器の実装

理想的には，ソースコードをそのまま与えるだけで，無害なバグを入れたソースコードを出力する変換器を提供することが望ましい．しかし，そのような変換器の実現には，通常，C 言語の構文解析プログラムなどの開発を伴い，開発コストは必ずしも小さくない．よって本研究では暫定的に，利用者がソースコードに変換支援のためのアノテーションを加えることを仮定して変換器を実装した．アノテーションは，#pragma refuge，#pragma dummy，#pragma overflow，#pragma free の 4 種類からなる．変換器は入力のソースコードを 1 行ずつ読み込み，基本的にはそのまま出力する．ただし，アノテーションを読み込んだ場合には，その種類に応じたコードを出力する．変換器は Python で記述されている．

```
#pragma refuge /* 退避用の配列の宣言場所 */

int main(void)
{
  char buf1[80], buf2[40];
  FILE *fp;
/* オーバーフローさせるバッファの宣言場所 */
#pragma dummy
  ...
/* オーバーフローの処理を追加する場所 */
#pragma overflow
  fgets(buf1, 80, fp);
  strcpy(buf2, buf1);
  ...
/* 確保したバッファの解放を行う場所 */
#pragma free
}
```

図 3 アノテーションを追加したプログラム
Fig. 3 Sample program with annotations

図 3 は，図 1 のプログラムにアノテーションを加えたものである．データ退避先のメモリ領域を宣言すべき場所に #pragma refuge を加える．また，バッファオーバフローさせるダミーのバッファを宣言すべき場所に #pragma dummy を加える．変換器は #pragma refuge や #pragma dummy のアノテーションを読み込んだら，その場所に適切な変数宣言を出力する．変数名はランダムに生成する．退避先のメモリ領域のサイズは，バッファオーバフローで上書きされるデータのサイズよりも大きいものに自動的に決定される．

#pragma overflow というアノテーションが書かれた部分には，(1) データ退避の処理，(2) バッファオーバーフローを起こす処理，(3) データ回復の処理が，その順にひとまとめに出力される．(1) では，ダミーのバッファの先頭アドレスから始まる十分な長さのデータを，#pragma refuge の変換の際に宣言されたメモリ領域にコピーする．(2) では，ダミーのバッファに対して，そのサイズ以上のデータを書き込む処理を出力する．書き込むデータはランダムの文字列とする．(3) では，(1) と逆の処理を行う．各々に関して，スタック上バッファのオーバーフローとヒープ上バッファのオーバーフローの 2 種類のためのコードが出力される．

アノテーションを追加した関数の最後には #pragma free というアノテーションも追加しなければならない．変換器はこのアノテーションを読み込むとメモリ解放のためのコードを出力する．

4. 議　　論

本研究の方式をオープンソースソフトウェアへ適用する際には注意が必要である．オープンソースソフトウェアの意義の一つは，世界の人々によるコードの理解や改変がしやすい点にある．しかし，本方式による変換後のソースコードだけが配布される場合，コードの理解や改変は，より困難になる．現状では，変換がもたらすトレードオフを了解して用いる必要がある．

別の注意は，変換後のコードのどの部分が元々あったもので，どの部分が自動挿入されたものかを，現状では，認識しやすいことである．変換方式を熟知した攻撃者は，挿入された部分を取り除く逆変換器を構築できる．この問題については，攻撃と防御が限りない循環に陥るため完全な解決は難しいが，逆変換を難しくすることには実践的には意味がある可能性がある．

5. 実　　験

Halebi が加えた無害なバグを，既存の脆弱性検査ツールが検出するかどうかを調べる実験を行った．実験には，Windows 7 (32 bit) を用いた．アプリケーションとして圧縮用プログラム MiniGzip（`http://free.pjc.co.jp/Zlib/`）を用いた．

まず，Halebi による変換後のプログラムに対して，Visual Studio 2010 Ultimate による静的コード分析を適用した．結果の一部を**図 4** に示す．変換によって加わった処理に対してバッファオーバーフロー（オーバラン）の明確な警告が出ている．次に，変換後のプログラムを RATS により検査した．結果の一部を**図 5** に示す．RATS では固定長バッファの宣言に警告を出す．Halebi は固定長バッファの宣言を加えるため，それに対する警告が出ている．また，データの退避，書き戻し処理のために加えた `memcpy` 関数の呼び出しに対する警告も出されている．

なお，Windows 8 (64 bit) 上で動く Visual Studio 2012 Ultimate でも実験を行い，スタックとヒープの両方のバッファに関して，範囲外を越えた読み出しと書き込みに対して警告が出るという同様の結果を得た．

6. まとめと今後の課題

C 言語のプログラムに，バッファオーバーフローを起こすが制御を奪われることがないような無害なバッファオーバーフロー処理を追加する変換を提案した．

今後の課題として以下の事項が挙げられる．第一には，ソースプログラムにアノテーションを加えなくとも変換ができるようにすることである．第二には，バッファオーバフローに限らない様々な種類のバグを含ませることができるようにすることである．第三には，難読化の効果や実行時オーバヘッドを定量的に評価することである．

```
...: warning C6203: スタックでないバッファー
 'cxEHXB13SCj7P' の 'strcpy' への呼び出し内
 でのバッファー オーバーランです: 長さ '17' は
 バッファー サイズ '8' を超えています
...: warning C6202: 'crE5v2f7ygXCV' のバッ
 ファー オーバーランです. 'memcpy' への呼び出
 しでスタックが割り当てられた可能性があります:
 長さ '246' はバッファー サイズ '8' を超えて
 います
...
...: warning C4789: メモリ コピーのターゲット
が小さすぎます
...: warning C4789: メモリ コピーのターゲット
が小さすぎます
```

図 4　変換後のプログラムを Visual Studio が検査した結果
Fig. 4　Output by Visual Studio when checking the sample program after transformation

```
sample.c:9: High: fixed size local buffer
sample.c:10: High: fixed size local buffer
Extra care should be taken to ensure that
character arrays that are allocated on the
stack are used safely. They are prime
targets for buffer overflow attacks.
sample.c:5: Low: fixed size global buffer
sample.c:6: Low: fixed size global buffer
Extra care should be taken to ensure that
character arrays that are allocated with a
static size are used safely. ...
sample.c:15: Low: memcpy
...
sample.c:32: Low: memcpy
Double check that your buffer is as big as
you specify.  ...
```

図 5　変換後のプログラムを RATS が検査した結果
Fig. 5　Output by RATS when checking the sample program after transformation

謝辞　静岡理工科大学の大石和臣氏，岡山大学の山内利宏氏，電気通信大学の高橋一志氏から本研究に対して有益な助言をいただいた．

参 考 文 献

1) Coverity SAVE, `http://www.coverity.com/products/coverity-save.html`.
2) Flawfinder, `http://www.dwheeler.com/flawfinder/`.
3) RATS, `https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml`.
4) Splint, `http://www.splint.org/`.
5) Viega, J., Bloch, J. T., Kohno, Y. and McGraw, G.: ITS4: A Static Vulnerability Scanner for C and C++ Code, *Proc. of ACSAC 2000* (2000).