Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

無害なバグを大量に含ませるプログラム変換器

大山恵弘[†1]　甲斐朋希[†1]

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，悪意の者が，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．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 Transformer for Injecting Numerous Harmless Bugs

Yoshihiro Oyama [†1] and Tomoki Kai[†1]

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. For example, the transformation adds buggy code causing a buffer overflow in which an attacker can never take the control. Since a vulnerability scanner outputs plenty of warnings against the injected bugs, actual vulnerabilities, if any, become more inconspicuous.

1. はじめに

マルウェアによるソフトウェアへの攻撃を許してしまう主な原因は，ソフトウェアに存在する脆弱性である．ソフトウェアの脆弱性を早期に発見し，修正するために，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[2)–6)]．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．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，攻撃を成功させる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，ソースプログラムを変換器によって変換し，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プログラム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，悪意の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 言語のソースプログラムを受け取り，無害なバグを含ませた C 言語のソースプログラムを出力する変換器を実装した．さらに，実験により，変換前と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や，変換によって加わる実行時間オーバヘッドを調査した．

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研究することを

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

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

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．本論文ではその変換について述べる．

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 (Rough Auditing Tool for Security)[4] は，C，C++，Perl，PHP，Python，Ruby などの言語のソースコードを検査することが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[6] 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，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り行われる．Flawfinder[3] 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 C 言語のライブラリ関数を発見し，プログラマに適切なアドバイスを表示する．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[5] 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．プログラマが C 言語のコメントの中に Splint 向けの指示を記述して解析を支援する点に特徴がある．Microsoft 社による Visual Studio 2010[7] の上位バージョンには「静的コード分析」と呼ばれる，バグの発見を支援する機能が備わっている．gcc コンパイラも，コンパイル時に静的にバグを検出する機能を提供している．例えば，コンパイルオプションに `-Warray-bounds` をつけると，範囲外の添字による配列アクセスに対して警告が出される．Coverity Static Analysis[2] は制御フローを意識した静的解析によって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

提案する変換器は，C 言語のソースプログラムを入力として受け取り，そのプログラムの動作を変えないようにしつつも，中に大量の無害なバグを加えた C 言語のソースプログラムを出力する．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，この変換後ソースプログラム，もしくは，それをビルドした実行型バイナリを配布する．変換前のプログラムは配布しない．

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，既に変換器に実装されている方法について説明する．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．このプログラムは真のバッファオーバフロー脆弱性が非常に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．本研究の変換器を使えば，この脆弱性を目立たなくさせることができる．

このプログラムを変換器で変換すると，**図 2** のプログラムが出力される．変換後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使われる．関数の中程の 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．スタック上とヒープ上の 2 種類バッファのオーバフローの処理を加えるのは，様々な脆弱性のパ

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

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

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

ターンを作るためである．バッファオーバフロー処理の前後には，上書きされるメモリ領域のデータを退避，書き戻しするための処理を出力する．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，ソースプログラムをそのまま与えると，無害なバグを入れたプログラムを出力する変換器を提供することが望ましい．ただし，そのような変換器の実現は，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，利用者がソースプログラム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．変換器は入力として与えられたソースプログラムを 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，その種類に応じた処理を出力する．変換器は Python で記述されている．

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．

```
char yPiwiJ2fkGe[224];
char h8Lpa4Z2TyVs[240];
int main(void)
{
  char buf1[80], buf2[40];
  FILE *fp;
  char JLHvPkBWeedy3d[8]; /* スタック上バッファ */
  char *NQzo1kmkHaJlU7 = (char *)malloc(8); /* ヒープ上バッファ */
  memcpy(yPiwiJ2fkGe, JLHvPkBWeedy3d, 222); /* データの退避 */
  memcpy(h8Lpa4Z2TyVs, NQzo1kmkHaJlU7, 238); /* データの退避 */
  strcpy(JLHvPkBWeedy3d, "XTyZE53Ul8Pk4"); /* スタック上バッファのオーバフロー */
  strcpy(NQzo1kmkHaJlU7, "Duka5y3cuPSS"); /* ヒープ上バッファのオーバフロー */
  memcpy(JLHvPkBWeedy3d, yPiwiJ2fkGe, 222); /* 退避したデータの書き戻し */
  memcpy(NQzo1kmkHaJlU7, h8Lpa4Z2TyVs, 238); /* 退避したデータの書き戻し */
  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

`#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．脆弱性検査ツールは (2) と (3) の処理をバッファオーバフローとして検出する可能性がある．

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

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

```
#pragma refuge /* 退避用の配列の宣言場所 */

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

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

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. 実　　験

4.1 オーバーヘッド

提案した変換器を用いて実験を行った．実験環境は，CPU が Intel Core 2 Duo 2.13 GHz，メモリが 2.0 GB の PC である．OS には Windows 7 Professional (32 bit) を用いた．

まず，変換後のプログラムでは元のプログラムと比べてどれだけ実行時間が増加するかを測定した．この実験では gzip と同様の動作を行う MiniGzip というプログラム（`http://free.pjc.co.jp/Zlib/`）を用いた．MiniGzip のソースプログラムを変換器で変換してできたソースプログラムをビルドして実行可能バイナリを作った．MiniGzip を用いて 100MB のファイルを圧縮，展開する動作をそれぞれ 15 回ずつ行い，その実行時間の平均を計測した．なお，この実験では，変換後のプログラムは 1 回の実行中に無害なバッファオーバーフロー処理を 1000 回実行した．結果を**表 1** に示す．無害なバッファオーバーフロー処理を 1000 回実行しても，元のプログラムと比べて実行時間に大きな差はなく，逆に実行時間は減少した．減少した原因は現在調査中である．

表 1 MiniGzip の実行時間
Table 1 Execution times of MiniGzip

	圧縮にかかった時間 [ms]	展開にかかった時間 [ms]
変換前	6544	874
変換後	6523	841

4.2 脆弱性検査ツールの使用

本研究の変換器が加えた無害なバグを脆弱性検査ツールが検出することを確認する実験を行った．用いた脆弱性検査ツールは Visual Studio 2010 Ultimate[7] と RATS[4] である．

まず，変換を行う前のプログラムに対して，Visual Studio による静的コード分析を適用した．分析結果として出力された結果の一部を**図 4** に示す．`strcpy` 関数など使い方によってはバッファオーバーフローを起こす関数を使用している箇所を挙げて，別の関数の利用を促していることがわかる．しかし，バッファオーバーフローが起こると明確に警告する出力は見当たらなかった．

次に，変換後のプログラムに対して静的コード分析を適用した．結果の一部を**図 5** に示す．変換前から出ていた警告に加えて，変換器が加えた処理に対する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．なお，増えた警告の数は変換器が加えた `#pragma overflow` の数に比例した．本実験において，図 4 に示した警告は英語で出力されたが，図 5 に示した警告は日本語で出力された．英語版 OS の環境では，ともに英語の出力になると予想される．

さらに，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 6** に示す．RATS の脆弱性データベースに登録された C 言語の関数には 3 つの危険度が定められているが，ここではすべての危険度の関数に対する検査を行うように指定した．結果からわかるように，RATS では固定長バッファの宣言に警告を出す．変換器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変換器が加えた `memcpy` 関数の呼び出しに対する警告も出されている．その数は変換器が追加した `#pragma overflow` の数に比例している．本研究の変換により，RATS に大量の警告を出させることができたことは確認した．ただ，RATS に大量の警告を出させるには，提案方式が行うほどの複雑な変換は必要がなく，より単純なツールでも十分である．

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

```
...
2>..\ZlibSrc\minigzip.c(203): warning C4996: 'strcpy': This function
 or variable may be unsafe. Consider using strcpy_s instead. To
 disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help
 for details.
2>..\ZlibSrc\minigzip.c(204): warning C4996: 'strcat': This function
 or variable may be unsafe. Consider using strcat_s instead. To
 disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help
 for details.
2>..\ZlibSrc\minigzip.c(206): warning C4996: 'fopen': This function
 or variable may be unsafe. Consider using fopen_s instead. To
 disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online
 help for details.
...
```

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

```
...
2>c:\users\kai\desktop\gzip1\zlibsrc\minigzip.c(469): 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
2>c:\users\kai\desktop\gzip1\zlibsrc\minigzip.c(470): 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
...
2>c:\users\kai\desktop\gzip1\zlibsrc\minigzip.c(302): warning C4789:
 メモリ コピーのターゲットが小さすぎます
2>c:\users\kai\desktop\gzip1\zlibsrc\minigzip.c(308): warning C4789:
 メモリ コピーのターゲットが小さすぎます
...
```

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

5. 議　　論

5.1 他の無害化方式

提案の変換器は，決まったパターンのバッファオーバーフローだけを繰り返し追加する．そのため現段階では，本研究の変換を意識した攻撃者が解析すれば，偽の脆弱性またはそれに対する警告を簡単にフィルタリングできる可能性がある．そこで，さらに多くの無害化方式を加えて，真の脆弱性を発見されにくくすることを検討している．

バッファオーバーフローを無害なものにする手段として，上で述べた退避と書き戻しに基づく方法の他に，2 つを検討している．第一は，バッファオーバーフローさせる配列の後ろに別のダミーの配列を配置する方法である．バッファオーバフローにより，配列の境界を越えたメモリ領域を壊したとしても，そこにはダミーの配列が存在するだけなので，プログラムの動作を変えることはない．しかし，極めてバグに見え易いメモリアクセスをプログラムが行うため，脆弱性検査ツールはバグとして検出する可能性がある．この方法を確実に実現するには，スタックフレームにおける配列のレイアウトを制御する必要がある．よって，本研究のようなソースレベル変換器だけでは，無害であることを保証できない可能性がある．

第二は，`setjmp` 関数と `longjmp` 関数を用いる方法である．この方法では，ダミーの関数を新たに定義し，呼び出す．ダミーの関数を呼び出す前に，`setjmp` 関数によって実行状態を保存しておく．ダミーの関数の中では，ダミーのバッファをスタックから確保し，バッファオーバフローによってリターンアドレス領域を破壊する．その後，ダミーの関数の中で `longjmp` 関数を呼び出し，`setjmp` 関数を呼び出した場所に復帰する．よって，リターンアドレス領域に上書きされた値は用いられず，バッファオーバフロー攻撃は成功しない．

5.2 プログラムの意味

本研究の変換は厳密にはプログラムの意味を変える．しかし，現状では，メモリ上のデータを無駄にコピーする処理を加えているだけなので，プログラムの外部からは，その変化を観測することが難しい．

C 言語の仕様では，確保されたメモリ領域の外をアクセスした場合の動作は未定義である．変換後のプログラムは，確保されたメモリ領域の外のデータを読み書きするが，仕様上は，その読み書きの結果については，何も保証されない．現実には，多くのコンパイラでコンパイルされたコードにおいて，確保されていない領域の読み書きの結果は予測可能なものになっている．本研究の方式はその性質に依存している．

データの待避と書き戻しによってバッファオーバフローを無害化する方式の実装や利用にあたっては，以下の点に注意する必要がある．第一には，加えた処理がセグメント違反エラーやメモリ不足エラーを引き起こす場合がある点である．バッファオーバフローにおいて

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

```
...
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. This appears to be a
global allocation and is less dangerous than a similar one on the
stack. Extra caution is still advised, however.
sample.c:15: Low: memcpy
...
sample.c:32: Low: memcpy
Double check that your buffer is as big as you specify.
When using functions that accept a number n of bytes to copy, such
as strncpy, be aware that if the dest buffer size = n it may not
NULL-terminate the string.
```

図 6 変換後のプログラムを RATS が検査した結果
Fig. 6 Output by RATS when checking the sample program after transformation

書き込むデータのサイズを大きく設定すると，確保されている（ページが割り当てられている）メモリ領域の外に書き込む可能性が生じる．現在は，オーバフローの幅を小さく抑えることで，この問題ができるだけ顕在化しないようにしている．また，退避用に大きなバッファを多数確保すると，メモリ不足を引き起こす可能性がある．この問題は，加えるバグの数を非常に大きくしない限り顕在化しないと考えられる．

第二には，バッファオーバフローにおいて書き込むデータのサイズを大きく設定すると，書き戻し処理の実行で利用されるメモリ領域さえも破壊し，データを正常に書き戻せなくなる場合がある点である．現在は，上で述べたものと同じく，オーバフローの幅を小さく抑えるという対策をとっている．

第三には，バッファオーバフロー後に元のプログラムのコードを実行してから書き戻し処理を実行するという変換方式を採用した場合に，バッファオーバフローで破壊されたデータをプログラムが使用する場合がある点である．現在は，バッファオーバフロー処理の直後に書き戻し処理を出力してこの問題を避けている．しかし，バッファオーバフローと書き戻しの処理はできるだけ離れた位置に出力したほうが解析が難しくなるため，現在の方式が最適かどうかは一概には言えない．

5.3 ダミーの変数の名前

現在はダミーの変数の名前を乱数で自動生成している．この方式には，自動生成された変数と，元から存在した変数を容易に区別できるという欠点がある．将来的には，ダミーの変数に対して，意味があるかのようなもっともらしい名前を自動的につけることが望ましいと考えている．ソースプログラムに書かれた変数名を加工すれば，もっともらしい名前を効率的に自動生成することができると予想している．

6. 関 連 研 究

本研究はプログラムの解析を難しくするためのツールを提案している．一般にプログラムを読みにくくする操作は難読化（obfuscation）と呼ばれる．本研究の変換器は難読化ツールの一種であるとみなせる．難読化ツールはこれまでに多く提案されている．例えば，COBF[1)] は C/C++用の難読化ツールである．COBF は変数やキーワードなどを別名に置き換えることによってプログラムを難読化する．プリプロセッサが元の名前に戻すことにより，プログラムを正しく実行する．攻撃者による脆弱性発見を妨害することは，COBF のような難読化ツールによっても可能である．人間がプ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，COBF を含む多くの難読化ツールはプログラムの意味を全く変えないので，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．

7. 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．その変換を実行する変換器を設計，実装し，実験を行った．

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．現在の変換器では，無害なバッファオーバーフロー処理を大量に含ませたい場合には，同じだけ大量の修正を元のプログラムに加えることが必要である．よって，利用に大きな手間がかかることがある．将来は，gcc などの既存の C コンパイラを改造し，ソースプログラムを修正しなくとも変換を行えるツー

Vol.2011-CSEC-54 No.36
Vol.2011-SPT-1 No.36
2011/7/13

ルを作れれば理想的であると考えている．

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．現在の変換器では，対象が C 言語プログラムに限られており，かつ，含ませる脆弱性もバッファオーバフロー脆弱性に限られているという点で，妨害できる攻撃者の範囲が限定される．近年はバッファオーバフロー攻撃と同等またはそれ以上に，他の攻撃の脅威が大きくなっている．例えばクロスサイトスクリプティング攻撃や SQL インジェクション攻撃といった Web アプリケーションへの攻撃が深刻である．それらの攻撃が可能であるかのように見せかける無害なバグを入れられれば，より広い範囲の攻撃者を妨害することができると考えられる．

参 考 文 献

1) COBF, http://www.plexaure.de/cobf/.
2) Coverity Static Analysis, http://www.coverity.com/products/static-analysis.html.
3) Flawfinder, http://www.dwheeler.com/flawfinder/.
4) RATS, https://www.fortify.com/ssa-elements/threat-intelligence/rats.html.
5) Splint, http://www.splint.org/.
6) Viega, J., Bloch, J.T., Kohno, Y. and McGraw, G.: ITS4: A Static Vulnerability Scanner for C and C++ Code, *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000)* (2000).
7) Visual Studio, http://www.microsoft.com/japan/visualstudio.