Vol.2009-OS-112 No.8
2009/8/6

Taint Analysis によるスパイウェア検知手法の回避

落合 淳[†1] 嶋村 誠[†1] 河野 健二[†1,†2]

現在，ユーザの個人情報を不正に収集するスパイウェアを検出する手法として *Taint Analysis* が注目されている．Taint Analysis では，パスワードなどのデータに *taint* と呼ばれる情報を付加し，taint 情報を持つデータが操作されたときに伝播規則に基づいて taint 情報を伝播する．そして，taint 情報を含むデータが外部に流出した場合にスパイウェアとして検出する．既存のスパイウェアは Taint Analysis を回避していない．そのため，Taint Analysis はスパイウェアが回避手法を用いた場合を考慮していない．本論文では，Taint Analysis の回避手法への耐性を向上させるため，既存の Taint Analysis に対する回避手法を示し，taint 情報の伝播規則の問題点を指摘する．回避手法を検証するため，一般的な伝播規則を用いた Taint Analysis を実装し実験を行った．実験の結果，回避コードを組み込むことで，Taint Analysis による検出ができなくなることを確認した．これにより，taint 情報の伝播規則を改良する必要があることを示した．

Evasion of Taint Analysis for Spyware Detection

Atsushi Ochiai,[†1] Makoto Shimamura[†1]
and Kenji Kono[†1,†2]

Taint analysis is a promising approach to detecting malicious behavior of a leaking users' sensitive data. In taint analysis, we label data as *tainted* such as password. When a program processes tainted data, we propagate taint attribute in the program, based on a given taint propagation rule. If tainted data is output to untrusted sources, we can recognize that the computer system is compromised. Evasion techniques to taint analysis have not been explored adequately. This means that taint propagation rules are not studied as well. To make taint analysis more powerful, we study current tainting rules and point out the weakness of the rules by using our evasion techniques. We implemented a taint tracker with the well-used propagation rule. Our experimental results demonstrate that the well-used taint propagation rule does not detect the data leak of test programs including our evasion techniques.

1. はじめに

現在，スパイウェアがインターネットユーザにとって脅威となっている[1),2)]．スパイウェアとは，悪意のあるソフトウェアの一種であり，ユーザのウェブページ閲覧履歴や入力したパスワード等をユーザの許可無く収集する．ウィルスやワームなどがユーザに対して目に見える破壊活動を行うのと比べて，スパイウェアはユーザに気づかれないように活動を行う．このため，コンピュータがスパイウェアに感染していたとしても分かりにくいことが多い．スパイウェアを検出する方法として，プログラム中のバイト列が既に発見されているスパイウェアと一致するか調べる手法が広く使われている[3),4)]．しかし，この手法は検出するスパイウェア毎にバイト列を用意しておく必要があり，新たに作成されたスパイウェアやバイト列に変更が加えられた亜種のスパイウェアを検出できない．

スパイウェアを検出する解析手法の一つとして，プログラム中で着目したいデータを追跡してプログラムを解析する Taint Analysis という手法が提案されている．Taint Analysis では，着目するデータに taint と呼ばれる情報を付加して，taint 情報を持つデータが他の変数を書き換えた際にその変数に taint 情報を伝播させてデータの流れを追跡する．そして taint 情報を持つデータがプログラム内で望まれない処理に利用されるか調べる．例えば Egele らの手法[5)] では，ユーザが閲覧したウェブページの URL データに taint 情報を与えて，意図していない外部のサーバに taint 情報を含んだデータが送信されるか調べる．Taint Analysis では個々のスパイウェアに関する情報を用いずに解析を行うため，既存のスパイウェアのみならず，新種や亜種のスパイウェアを検出できると考えられている．この点から，Taint Analysis はスパイウェアの検知手法として有用であると考えられている．

しかし既存の Taint Analysis は，スパイウェアが回避方法を用いることを考慮していない．今後 Taint Analysis を想定したスパイウェアの作成がされた場合，様々な回避コードが組み込まれたスパイウェアが生成されることが想定される．既存の Taint Analysis は回避方法に対しての耐性を持たないため，Taint Analysis による解析は容易に無力化されてしまう．回避方法への耐性が必要である一方，どのような方法によって Taint Analysis が

†1 慶應義塾大学 理工学部 情報工学科
Department of Information and Computer Science, Keio University
†2 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

回避されるかはあまり知られておらず，回避方法への対策もほとんどされていない．Taint Analysis によるスパイウェアの検出精度を高めるためには，Taint Analysis の回避手法について検討し，既存の伝播規則を用いた検知システムの問題点を明らかにする必要がある．

そこで本論文では，既存の Taint Analysis を用いたスパイウェア検知システムにおける taint 情報の伝播規則の問題点を指摘する．現在の不十分な伝播規則に対して，どのような回避手法を用いると taint 情報が伝播しなくなり解析できなくなるのか，具体的な回避コードを示し議論を行う．不十分な taint 情報の伝播規則によって問題となる回避手法として

- 代入先に taint 情報を持つ値を用いた回避手法
- 既知の回避方法への対策では不十分な点に着目した回避手法
- 関数呼び出し時の挙動に基づいた回避手法

を示す．

本研究で示す回避手法を検証するために，一般的な taint 情報の伝播規則を用いた Taint Analysis を実装した．実験として，考案した回避コードを組み込んだ cat プログラムと，組み込んでいない cat プログラムをそれぞれ与えて，read システムコールで読み込まれるデータに taint 情報を付加し，write システムコールで書き出されるデータに taint 情報が含まれるか調べた．解析の結果，回避手法を用いていないプログラムは taint 情報を含んだデータの標準出力への書き込みを検知することができたが，回避手法を用いたプログラムは標準出力への書き込みを検知することができなかった．これにより既存の伝播規則では taint 情報の伝播が途切れる事が分かった．従って，既存の伝播規則はデータの流出を検知するためには不十分であることが示された．

以下，2 章では Taint Analysis の手法，及び関連研究について説明する．3 章では taint 情報の伝播規則の不十分な点を指摘し，問題点に基づいた回避手法について示す．4 章では実験および実験結果を示す．5 章では taint 情報の伝播規則の改良案について議論する．6 章では本論文のまとめと今後の課題について述べる．

2. Taint Analysis

2.1 解析方法

Taint Analysis は着目したいデータに taint 情報を与えて，プログラム中の taint 情報の伝播を追跡することによりプログラムを解析する手法である．多くの Taint Analysis ではデータ 1 バイト毎に taint 情報を与えて解析する．実行時に動的に taint 情報の伝播処理を行うものを特に Dynamic Taint Analysis と呼び，解析したいプログラムのソースコードを

```
x^tainted = input();   // (1) 着目したいデータに taint 情報を与える
y^tainted = x^tainted; // (2) 着目したデータに依存して書き換えられるデータに taint 情報を
                          伝播させる
send(y^tainted);       // (3) 外部に送信されるデータに taint 情報が含まれるか検査する
```

図 1 Taint Analysis の基本処理
Fig. 1 Basic process of taint analysis

用いることなくデータの流れを追うことができる[6]．Dynamic Taint Analysis は対象となるプログラムのコンパイル時に特別な変更を加えずにプログラムを解析できる．一方，静的に Taint Analysis を行うシステムもある[7]．しかし，静的な Taint Analysis は，スパイウェア開発者によって暗号化や難読化されたプログラムを解析するのが困難であり，スパイウェアの検出を行う目的には向かない．そのため本論文では，Taint Analysis として Dynamic Taint Analysis に着目する．

ユーザの入力を収集し外部に送信するスパイウェアを検出する場合，**図 1** で示すように Taint Analysis では，(1) 着目したいデータに taint 情報を与える，(2) 着目したデータに依存して書き換えられるデータに taint 情報を伝播させる，(3) 外部に送信されるデータに taint 情報が含まれるか検査する，3 段階の処理を行う． taint 情報の伝播は機械語レベルで処理し，多くの Taint Analysis を用いた検知システムでは，TaintCheck[6] で用いられた以下の taint 情報の伝播規則が使用されている．

- データ移動命令（LOAD, STORE, MOVE, PUSH, POP, etc.）を実行する際，移動元のデータが taint 情報を保持していた場合，移動先のデータに taint 情報を与える．
- 演算命令（ADD, SUB, XOR, etc.）を実行する際，オペランドのうち 1 つでも taint 情報を保持しているものがある場合，演算結果に taint 情報を与える．
- データ移動命令，演算命令において，代入元のデータが taint 情報を保持していない場合，代入先のデータの taint 情報を取り除く．
- 定数は taint 情報を保持しない．
- “xor eax, eax” のように，オペランドに依存しない演算結果が得られる場合，代入先から taint 情報を取り除く．

2.2 現状と問題点

現在の Taint Analysis を用いる検知システムは，現存するスパイウェアを高い精度で検

```
1: x^tainted;              // 0, 1
2: if (x^tainted == 0)     y^untainted = 0;
3: else                    y^untainted = 1;
4: send(y^untainted);      // y(=x) が流出
```

図 2 条件分岐を用いた値のコピー
Fig. 2 Copying a value via a conditional branch

```
1: x^tainted;              // 0, 1, ..., 255
2: array[256]^untainted = {0, 1, ..., 255};
3: y^untainted = array[x^tainted]^untainted;
4: send(y^untainted);      // y(=x) が流出
```

図 3 変換テーブルを用いた値のコピー
Fig. 3 Copying a value via a translation table

出できる[5),8)]．スパイウェアの作成者は，プログラムの暗号化や難読化といった方法を用いて，スパイウェアが解析によって検出されるのを回避する．これに対して，Taint Analysis ではデータの流れを追いかけて解析を行うため，暗号化や難読化されたスパイウェアへの耐性が高く，既存のスパイウェアを精度良く検出できる．

しかし Taint Analysis には，taint 情報が伝播しなくなることによって，着目するデータを追跡できなくなる問題があることが知られている．Taint Analysis は代入や演算によって taint 情報を伝播させるため，値のコピーが直接の代入を行う命令や演算命令によらずに行われた場合，taint 情報の伝播は途絶える．**図 2** のコードで示される制御構造によって書き換わる値が変わる制御フローに基づいた依存性や，**図 3** のように変換テーブルを用いた処理があると，TaintCheck で用いた taint 情報の伝播規則では追跡できなくなることが知られている． これまでにこうした回避方法が発見されることによって，Taint Analysis の問題点が指摘されてきた．図 2 と図 3 で示される回避方法に対しては，既に taint 情報の伝播規則の追加がなされてデータの追跡ができるようになっている[5)]．一方で，依然あらゆる制御構造に対して taint 情報が正しく伝播できることは示されていない．そのため，スパイウェアの開発者は新たな Taint Analysis の回避方法を作成する可能性がある．

2.3 関連研究 - 検知システム

Egele らの手法[5)] では，閲覧したウェブページの URL データなどに taint 情報を与えて，BHO（Brower Helper Object）の挙動を解析することで，悪意のある BHO を検出する．このシステムでは，図 2 で示した条件分岐を用いた制御構造による回避方法と，図 3 で示した変換テーブルを用いた回避方法への対策を行っている．条件分岐を用いた制御構造による回避方法への解決策として，taint 情報を持つ値が条件式に用いられた場合に，制御ブロック内で変更されるすべての変数に taint 情報を与えることを提案している．例えば，**図 4** で示す制御構造では，制御ブロック内で定数が代入される変数 `clean` は，taint 情報が与えられる．また，変換テーブルを用いた回避方法に対しては，taint 情報を含んだデータを用いたアドレス計算が行われた際に，出力結果に taint 情報を伝播させる処理を追加している．この処理により，図 3 と同様のプログラムを解析すると，**図 5** で示すように taint 情報が伝播するようになる．

```
1: if (t == 'a')
2:     clean = 'a';
3: else {
4:     if (t == 'b')
5:         clean = 'b';
6:     else
7:         clean = 'c';
8: }
```

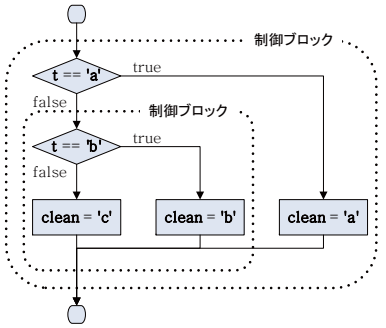

図 4 条件分岐を用いた回避方法への対策
Fig. 4 Countermeasure against an evasion technique that uses conditional branches

```
1: x^tainted;              // 0, 1, ..., 255
2: array[256]^untainted = {0, 1, ..., 255};
3: y^tainted = array[x^tainted]^tainted;
4: send(y^tainted);        // y(=x) が流出
```

図 5 変換テーブルを用いた回避方法への対策
Fig. 5 Countermeasure against an evasion technique that uses a translation table

Panorama[8)] は，ユーザのキー入力やファイルシステムの情報等に taint 情報を与えて，キー入力を収集するキーロガーや，ディレクトリ情報を改ざんし存在を隠蔽して悪意のある活動を支援するルートキットを検出するシステムである．taint 情報が付加されたデータの OS（Operating System）内での流れを解析することで，そのようなデータが本来与えられるべきでないプロセスに与えられるか調べる．不正なデータの流れを検知した場合，解析したプログラムは悪意のあるソフトウェアとして検出される．

2.4 関連研究 - 回避方法

Cavallaro ら[9)] は，Egele らの手法や Panorama による解析に対する回避方法を示した．第一に，Egele らの手法で用いられる制御ブロック内で書き換えられる変数に taint 情報を与える処理によって，一般的で安全な BHO に対しても多くの警告が出ることを指摘した．

これにより，Egele らが提案した図 4 のような回避方法への対策は有効ではないことを述べた．第二に，共有メモリからデータを読み出すことにより，taint 情報の付加を回避する方法を示した．この回避方法では，taint 情報が付加されないことによって解析ができなくなる．第三に，マルチスレッド環境下における競合状態を利用した回避方法を示した．taint 情報を持つデータの代入と，taint 情報を持たないデータの代入が複数のスレッドで競合して起こり得るシステムでは，taint 情報が正しく伝播しない場合があることを述べた．

3. 回避手法の考案

3.1 研究目的

本論文は，Taint Analysis を用いるスパイウェア検知システムに対する回避手法を議論することで，現在広く用いられている taint 情報の伝播規則の問題点を指摘することを目的とする．既知の伝播規則ではどのような回避手法を用いると taint 情報が途絶えて解析が行えなくなるのか，具体的な回避コードを用いて示す．これにより，既存の伝播規則はスパイウェア検知システムに用いるためには不十分であることを明らかにする．そして，回避手法に対する耐性を向上させるためには，伝播規則の改良が必要であることを示す．

3.2 対象とする Taint Analysis

本論文では，2.1 節で述べた taint 情報の伝播規則，及び以下の伝播規則を用いた，スパイウェア検知を行う Taint Analysis を対象とする．

- taint 情報を用いたアドレス計算が行われた際に，出力結果に taint 情報を伝播させる．
- taint 情報を持つ値が条件式に用いられた場合に，実行される制御ブロック内で書き換えられる全てのデータに taint 情報を与える．

また，taint 情報の与え方に依存しない包括的な Taint Analysis を対象とする．そのため，着目したいデータへの taint 情報の付加方法や検査方法は問わない．

3.3 考案する回避手法

本論文では，**図 6** で示すようにプログラム中で値をコピーした際に taint 情報が取り除かれて，taint 情報の伝播が途絶えることによる回避手法を考案する． 伝播処理に着目した回避手法は，全てのスパイウェアが組み込むことのできる回避手法であり，Taint Analysis を用いた全てのスパイウェア検知システムの脅威となる．一方で Cavallaro らが示したような，共有メモリからデータを読み出すことで taint 情報の付加を回避する方法や，競合状態を利用した回避方法は，検知システムの実装方法に依存する問題であるため，本論文では扱わない．

```
1: x^tainted;                              // 0, 1, ..., 255
2: y^untainted = launder(x^tainted);       // x の値を y にコピーし，taint 情報を取り除く
3: send(y^untainted);                      // y(=x) が流出
```

図 6 taint 情報の伝播が途絶える回避手法
Fig. 6 Evasion techniques to prevent propagation of taint-tags

```
1: for (i = 0; i < 256; i++)
2:     buf[i] = C;
3: buf[x] = '\0';
4: y = strlen(buf);
```

図 7 回避コード – 代入先に taint 情報を持った値を与える
Fig. 7 Evasion code – using a tainted value in assigned target

```
1: unsigned char strlen(const char *buf) {
2:     int i;
3:     for (i = 0; buf[i]; i++) ;
4:     return i;
5: }
```

図 8 strlen 関数の実装例
Fig. 8 Simple implementation of strlen function

Taint Analysis に対する回避手法が満たす性質は 2 点あり，第一にどのようにして値を保持・コピーするか，第二にどのようにして taint 情報を伝播させないかである．

3.3.1 代入先に taint 情報を持つ値を用いた回避手法

現在の Taint Analysis では，taint 情報を含むデータを用いて代入先のアドレス計算を行った場合は考慮されていない．このため，配列の添え字に taint 情報を持つデータを用いてその要素に定数を代入すると，taint 情報を持つデータに依存するデータ構造を taint 情報を取り除いた状態で作ることができる．

taint 情報を持つデータを代入先に与えることで，taint 情報を取り除きつつ値をコピーする手法を**図 7** に示す．ここで，変数 x は taint 情報を保持している値である． このプログラムでは，まず長さが 256 の配列 buf を用意して，要素を全て定数 C で初期化する．次に，3 行目で taint 情報を持つ変数 x を配列の添え字に与えて，その要素に対し定数'\0' を代入する．このとき，taint 情報を用いたアドレス計算が行われるため，配列 buf の x 番目の要素は taint 情報を持つものと評価される．しかし，他の変数に代入を行わないため taint 情報の伝播は起こらない．図 7 の 4 行目の処理で変数 x は使用されず，また配列 buf には taint 情報が含まれていない．このため，以後 taint 情報は検出されない．x 番目の要

Vol.2009-OS-112 No.8
2009/8/6

```
1: clean = 'c';
2: if (t == 'a')
3:     clean = 'a';
4: else if (t == 'b')
5:     clean = 'b';
```

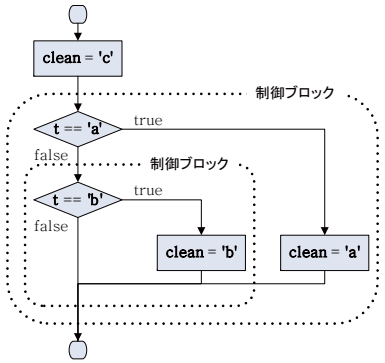

図 9 制御構造に入る前の値が考慮されていないため taint 情報が伝播しない
Fig. 9 Taint-tag doesn't propagate because the initial state is unconsidered

素のみが定数'\0' である配列を，標準ライブラリで用意されている `strlen` 関数に与えると，文字列の長さを求める要領で計算が行われる．**図 8** は `strlen` 関数の簡単な実装例である．`strlen` 関数は配列 `buf` で示される文字列の長さとして x と等しい値を返す．この結果，変数 y には taint 情報を含まない x と等しい値が代入される．

3.3.2 既知の回避方法への対策では不十分な点に着目した回避手法

Egele らの手法では，taint 情報を持つ値が条件式に用いられた場合に，制御ブロック内で書き換えられるデータに taint 情報を与えている．しかし，条件分岐によって実行されなかった側の制御ブロック内で書き換えられるデータに関しては言及していない．そのため，**図 9** で示す制御構造を用いると，図 4 で示したプログラムと等しい変数の状態が得られ，変数 `t` が文字 c と等しい場合，taint 情報が変数 `clean` に伝播しないことになる．

制御ブロックを通らないことで変数の状態が変わらない依存性を用いた回避手法を，**図 10** のコードを用いて示す．図 10 は図 7 と同様に，配列を用いて文字列の長さを求める処理を行い，taint 情報を伝播させない回避コードである． なお，この回避手法は，taint 情報を持った値で代入先のアドレス計算を行った場合に taint 情報が伝播する Taint Analysis を想定している．伝播させる方法に関しては 5 章で述べる．

図 10 で示すプログラムでは，まず長さが 257 の配列 `buf` を用意して，要素を全て定数'\0' で初期化する．次に，3 行目で taint 情報を含んだ変数 `x` を用いて配列の x + 1 番目の要素に定数 S を代入する．ここで，代入先のアドレスを taint 情報を持った値で計算することによって，taint 情報が x+1 番目に伝播する規則を用いた Taint Analysis を想定する．4 行

```
1: for (i = 0; i < 256 + 1; i++)
2:     buf[i] = '\0';
3: buf[x + 1] = S;  // 配列 buf の x+1 番目の要素に taint 情報 が伝播するシステムを想定する
4: for (i = 0; i < 256; i++)
5:     if (buf[i] == buf[i + 1])
6:         buf[i] = C;
7: y = strlen(buf);
```

図 10 回避コード – 値が書き換わらない依存性
Fig. 10 Evasion code – based on dependency of the initial state

表 1 図 10 のコードの実行経過
Table 1 Proccess of the code shown in Fig.10

		buf							
i の値	条件式	0	1	2	3	4	...	255	256
0	真	'\0'→ C	'\0'	'\0'	$\mathbf{S}^{tainted}$	'\0'	...	'\0'	'\0'
1	真	C	'\0'→ C	'\0'	$\mathbf{S}^{tainted}$	'\0'	...	'\0'	'\0'
2	偽	C	C	'\0'	$\mathbf{S}^{tainted}$	'\0'	...	'\0'	'\0'
3	偽	C	C	'\0'	$\mathbf{S}^{tainted}$	'\0'	...	'\0'	'\0'
4	真	C	C	'\0'	$\mathbf{S}^{tainted}$	'\0'→ C	...	'\0'	'\0'
⋮	⋮						⋱		
255	真	C	C	'\0'	$\mathbf{S}^{tainted}$	C	...	'\0'→ C	'\0'

目から 6 行目で，配列の隣接要素を比較して値が一致すれば値を定数 C に書き換える処理を繰り返す．変数 `x` の値が 2 である場合，繰り返し処理の経過の様子は**表 1** で示す通りになる． 表 1 では，i の値が 1 増えるごとに実行した際の，条件式の評価と配列の書き換わり方，及び taint 情報の伝播を示している．i の値が 0 の時の条件式は `buf[0]` と `buf[1]` が共に値が定数'\0' であるため真となり，`buf[0]` には定数 C が代入される．同様に i の値が 1 の時，`buf[1]` も定数 C に書き換えられる．i の値が 2 の時は `buf[2]` が定数'\0'，`buf[3]` が定数 S であるため，条件式は偽となり値は書き換えられない．ここで，`buf[3]` が taint 情報を含むため，伝播規則によって，実行される制御ブロック内で書き換えられる全てのデータに taint 情報が与えられる．しかし，この条件分岐によって実行される制御ブロックが無いため，taint 情報はどこにも与えられない．i の値が 3 の時も，値が 2 の時の処理と同様で，値は書き換えられず taint 情報も付加されない．繰り返し処理が終わった時点で，配列 `buf` の 0 個目と 1 個目の要素の値は定数 C，2 個目の値が定数'\0' であり，3 個目のみ

```
01: #define SP(ESP, N)     ((ESP) - 0x10 * (N) - (sizeof(long) * 1))
02: ...
03: asm volatile("movl %%esp, %0" : "=r" (esp));
04: recursive1(255 + 1);                // recursive1 関数を用いて 256 回再帰呼び出しをする
05: for (i = 0; i < 256; i++)
06:     stack_before[i] = *((unsigned long *) SP(esp, i));  // リターンアドレスを保存する
07: recursive2(x);                      // recursive2 関数を用いて x 回再帰呼び出しをする
08: for (i = 0; i < 256; i++)
09:     stack_after[i] = *((unsigned long *) SP(esp, i));  // リターンアドレスを保存する
10: for (i = 0; i < 256; i++)
11:     if (stack_before[i] == stack_after[i])
12:         break;
13 : y = i - 1;
```

図 11 回避コード – 関数呼び出し時の挙動を利用
Fig. 11 Evasion code – based on behavior of calling function

taint 情報を保持している．この状態で配列 `buf` を `strlen` 関数に与えると，2 が返される．`strlen` 関数は図 8 で示される通り，処理が配列 `buf` の 0 個目から 2 個目で完結するため，3 個目は評価されない．これによって，taint 情報は途切れることになる．

3.3.3 関数呼び出し時の挙動に基づいた回避手法

taint 情報を持った値に依存する関数呼び出しを行い，スタックを参照することで値が求められる点に基づいた回避手法を示す．既存の Taint Analysis は，関数呼び出し後の使用済みのスタックを考慮しておらず，スタックに対する特別な taint 情報の処理を行わない．スタック上には，関数呼び出しの処理に用いる引数やリターンアドレスが積まれる．通常，引数は push 命令や mov 命令によってスタックに積まれるため，元のデータが taint 情報を保持していた場合，taint 情報はスタック上の要素にも伝播する．しかし，リターンアドレスは呼び出し元によって決まる定数であり，また，call 命令によってスタックに積まれるため，既存の伝播規則では taint 情報を処理する対象でない．そのため，スタック上のリターンアドレスには taint 情報が与えられず，taint 情報が伝播することなく参照できる．

スタック上の要素を参照できることを基にして，引数に与える回数だけ再帰する関数を用いた回避手法を**図 11** を用いて示す． また，図 11 で示す回避コードによって，実行時にスタックが変わる様子を**図 12** に示す． 図 11 の 4 行目で再帰関数 `recursive1` に 256 を与えた際，スタックの状態は図 12 左側のようになる．この状態のスタック上に積まれた 256 個

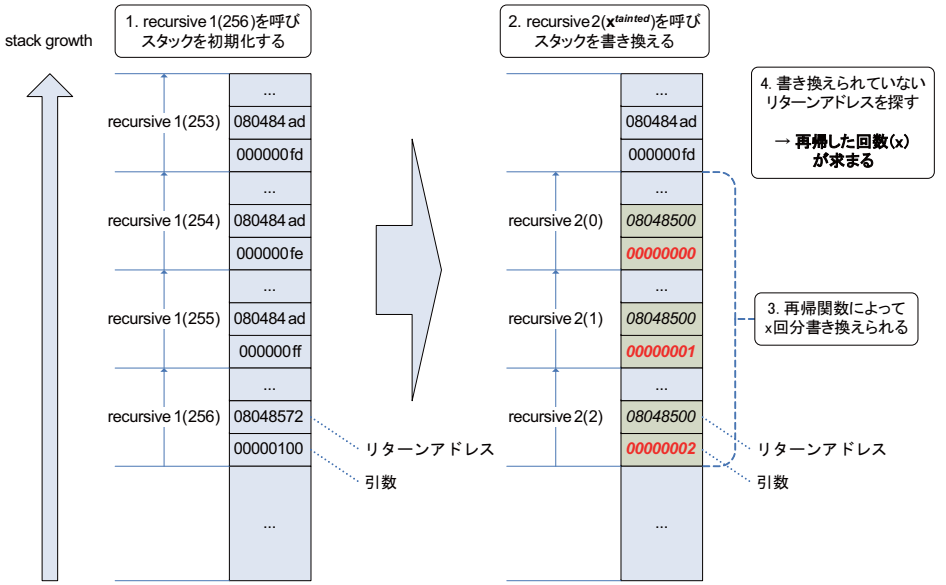

図 12 再帰関数によってスタックが書き換わる様子
Fig. 12 Process of rewritting the stack by recursive function

のリターンアドレスを呼び出し元で配列 `stack_before` として保存する．次に，7 行目で再帰関数 `recursive2` に変数 `x` を与えて x 回再帰させる．ここで，再帰関数はリターンアドレスが異なるように別の関数を用いている．x の値として 2 を与えた場合，スタックは図 12 右側のように書き換わる．図 12 では，値が書き換わったスタック上の要素を斜体で，taint 情報を保持している要素を太字で表している．図 12 で示す通り，スタック上のリターンアドレスや引数は再帰関数によって 3（2+1）回関数を呼び出した分書き換えられる．このとき，変数 `x` は taint 情報を持っているため，taint 情報がスタック上の引数に相当する要素に伝播する．しかし，リターンアドレスは call 命令によってスタックに積まれたため taint 情報が与えられない．再び，この状態のスタック上のリターンアドレスを配列 `stack_after` として保存する．保存された配列 `stack_before` と配列 `stack_after` には，taint 情報が含まれていない．配列 `stack_after` の書き換えられる範囲は，引数に与えた変数 `x` によって決まる．従って，配列 `stack_before` と配列 `stack_after` を比較することで x の値を求めることができる．このようにして，taint 情報を持たない x と等しい値を作ることができる．

```
1: while ((ch_in = getchar()) != EOF) {
2:     ch_out = launder(ch_in);
3:     putchar(ch_out);
4: }
```

図 13 検証用の cat プログラム
Fig. 13 Snippet of test cat program

表 2 解析結果
Table 2 Result of analysis

launder 関数	警告
値をそのまま返すコード	有り
図 7 で示したコード	無し
図 10 で示したコード	無し
図 11 で示したコード	無し

4. 実　　験

考案した回避手法を検証するために実験を行った．本章では，まず実験を行う環境を用意するために実装した Taint Analysis について説明する．そして，実装した Taint Analysis を用いて回避コードを組み込んだプログラムを解析した結果を示す．

4.1 Taint Analysis の実装

回避手法の検証を行うために，ハードウェアエミュレータである QEMU[10] を用いて Taint Analysis を行うシステムを実装した．QEMU 上のゲスト OS には Fedora Core 8 に Linux2.6.27 をインストールしたものを用いた．

本システムでは，対象とするプロセスの解析を開始する際に，ゲスト OS 上から QEMU に対して解析するプロセスの CR3 レジスタの値を通知する．CR3 レジスタの値はプロセス毎に固有の値をとるため[11]，QEMU は通知される CR3 レジスタの値を元に，プロセスを識別して解析を行う．

4.2 方　　法

検証用のプログラムとして，**図 13** で示す cat プログラムを用意した． 2 行目で呼び出す `launder` 関数として，それぞれの回避コード，あるいは値をそのまま返すコードを定義した．

read システムコールによって読み込まれるデータに対して taint 情報の付加を行った．そして，write システムコールによって書き出されるデータに taint 情報が含まれているを検査し，含まれていた場合に警告を発する処理を行った．

4.3 結　　果

実験を行った結果を**表 2** に示す．表 2 より，回避コードを組み込んでいないプログラムに対しては警告を発している．一方で，回避コードを組み込んだプログラムに対してはいずれも警告を発していないことが分かる．すなわち，回避コードを用いることで標準出力への書き込みの検知を回避している．

実験により，考案した回避手法である，

- 代入先に taint 情報を持つ値を用いた回避手法
- 既知の回避方法への対策では不十分な点に着目した回避手法
- 関数呼び出し時の挙動に基づいた回避手法

を用いると，既存の Taint Analysis で用いている伝播規則では，taint 情報が伝播しなくなることが分かった．これにより，既存の Taint Analysis が用いている taint 情報の伝播規則は，スパイウェアの検出に用いるためには不十分であることが示せた．

5. 考案した回避手法に対する議論

本論文で示した回避手法に対して，データの流出を検知できるようにするため，taint 情報の伝播規則の改良案を議論する．以下では，動的な解析に基づく改良案と，静的な解析に基づく改良案について示す．

5.1 動的な解析に基づく改良案

• taint 情報を持ったデータが代入先のアドレス計算に用いられた場合，代入元の taint 情報とアドレス計算を行ったデータの taint 情報を伝播させる

図 7 で示したコードのように，代入先で taint 情報を持ったデータをもとにしてアドレス計算を行った場合でも，taint 情報の伝播を考慮する必要がある．これには，代入元の taint 情報に加えて，アドレス計算を行ったデータの taint 情報を代入先に伝播させることで，taint 情報の追跡が可能になると考えられる．アドレス計算を行ったデータの taint 情報を伝播させることで，図 7 の 3 行目で配列 `buf` の x 番目の要素に taint 情報を与えることができる．この状態の配列を `strlen` 関数に与えると，taint 情報を含んだ x 番目の要素が条件式に用いられるため，伝播規則によって実行される制御ブロック内で書き換えられる全てのデータに taint 情報を与えられる．これにより，`strlen` 関数の返り値が taint 情報を持つようになる．従って，図 7 で示した回避手法は無効化できると考えられる．

• 関数で利用した範囲のスタックに taint 情報を ret 命令時に与える

図 11 で示した回避手法を解決するためには，関数呼び出し後のスタック上のリターンアドレスを参照した際に taint 情報を伝播させる必要がある．そのための改良案として，ret 命令時に関数で利用した範囲のスタックに taint 情報を付加することが考えられる．これにより，関数を呼んだ後にスタックを参照すると taint 情報が伝播するようになるため，図 11

で示したコードに対しても，taint 情報の伝播ができるようになる．よってスタックを利用した回避手法を無効化できるものと考えられる．

5.2 静的な解析に基づく改良案

• taint 情報を持つ値が条件式に用いられた場合に，いずれかの制御ブロックにおいて書き換えられ得る全てのデータに taint 情報を与える

図 10 で示したコードは，taint 情報を持つ値が条件式で評価された結果，制御ブロックに入らなかった場合の taint 情報の扱いが考慮されていない点に基づいた回避手法である．静的な Taint Analysis を行う STILL[7] では，制御構造によって変数に taint 情報が伝播する経路があれば，実行時にその経路を通らなくても，その変数に taint 情報を付加させる伝播規則を提案している．この伝播規則を用いると，図 10 の i の値が 2 の時の処理が taint 情報の伝播の対象となる．そのため，図 10 のコードは taint 情報が伝播するようになり，この回避手法を無効化できる．しかし，この手法では非常に多くの変数が taint 情報の伝播対象となってしまうため，誤検知が多発する可能性が高い．

6. おわりに

近年，Taint Analysis を用いたスパイウェア検知システムが提案されている．Taint Analysis を用いることで，重要なデータを外部に流出させるスパイウェアを高い精度で検出できる．しかし今後，スパイウェア開発者が回避コードを用いたスパイウェアを作成した場合，現在の Taint Analysis では検出できなくなる．現在，Taint Analysis の回避方法はあまり知られておらず，十分な対策は行われていない．

本論文では，既存の Taint Analysis を用いるスパイウェア検知システムにおける taint 情報の伝播規則の問題点を指摘した．既存の伝播規則では taint 情報の伝播が途切れる回避手法を考案し，具体的な回避コードとして示した．実験により，回避コードを組み込んだプログラムは，一般的に有効と考えられている伝播規則を用いた Taint Analysis では検出が行えなくなることが分かった．本研究によって，Taint Analysis をスパイウェア検知システムに用いるためには，taint 情報の伝播規則を改良する必要があることを示した．

今後の課題は，本論文で述べた改良案を Taint Analysis に実装すること，そして，改良した taint 情報の伝播規則を用いた Taint Analysis の有用性を評価することである．

参 考 文 献

1) Thompson, R.: Why Spyware Poses Multiple Threats to Security, *Communications of the ACM*, Vol.48, No.8, pp.41–43 (2005).
2) Saroiu, S., Gribble, S.D. and Levy, H.M.: Measurement and Analysis of Spyware in a University Environment, *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation*, pp.141–153 (2004).
3) Hackworth: Spybot Search & Destroy, http://www.safer-networking.org/ (2005).
4) Lavasoft: Ad-Aware, http://www.lavasoft.com/ (2005).
5) Egele, M., Kruegel, C., Kirda, E., Yin, H. and Song, D.: Dynamic Spyware Analysis, *Proceedings of the USENIX Annual Technical Conference*, pp.233–246 (2007).
6) Newsome, J. and Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, *Network and Distributed Systems Security Symposium* (2005).
7) Wang, X., Jhi, Y.-C., Zhu, S. and Liu, P.: STILL: Exploit Code Detection via Static Taint and Initialization Analyses, *Proceedings of the 24th Annual Computer Security Applications Conference*, pp.289–298 (2008). full version is available at "http://www.cse.psu.edu/~xinrwang/papers/still.pdf".
8) Yin, H., Song, D., Egele, M., Kruegel, C. and Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis, *Proceedings of the 14th ACM conference on Computer and communications security*, pp.116–127 (2007).
9) Cavallaro, L., Saxena, P. and Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment, *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp.143–163 (2008).
10) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp.41–46 (2005).
11) Jones, S.T., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment, *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pp.1–14 (2006).