2005－CSEC－29（2）
2005／5／19

セキュアコーディングと準形式設計モデル

倉光君郎[†]　村上　直[††]

ソフトウェア脆弱性の問題は、プログラマ個人のスキルの問題ではなく、ソフトウェア開発プロセスから対応すべきである。本研究では、セキュアコーディングの知見からセキュアソフトウェア設計を準形式的に定義し、その実用性と限界を論じる。

Semi-Formal Software Design for Secure Coding

KIMIO KURAMITSU [†] and TADASHI MURAKAMI [††]

Software vulnerabilities are a problem that the developers should treat with software lifecycle, including design, coding and testing. We use a semi-formal design model to define secure software behaviors. The paper will discuss how our model available is in terms of secure programming and software assurance.

1. はじめに

高度情報社会の深化にともない、ソフトウェア脆弱性が与える影響は広範囲な広がりをみせている。ソフトウェア開発者は、より安全で信頼性の高い（つまりセキュアな) ソフトウェアを作ることを社会から強く求められ、また同時にその実現に多大な関心を持たざる得なくなっている。

ソフトウェア脆弱性は、一般にコーディングレベルの問題に起因することが多い。現在では、過去の経験から様々なセキュアコーディングのテクニックが知られ、書籍[1)~3)]を通して広く啓蒙されている。そのため、開発マネージャは、プログラマ個人のスキルと努力によってこの問題を解決すべきと考えることが多い。しかし、この現場主義はソフトウェア製品の品質をまったく保証できない点で問題である。

近年、上流工程を含め、開発プロセス全体からセキュアソフトウェア開発へ取り組む必要性[4)]が主張され始めている。この考え方は、Correctness by Construction[5)]とも呼ばれる。従来のライフサイクル開発プロセスに対し、脅威分析やセキュリティテスト計画などを体系的に組み入れることで、現場のスキルへの依存度を下げている。これらの試みは、Microsoft Trusted Computing など、ある程度の成果は報告されているが、定性的な意味でのセキュアさを保証できない。

従来、定性的な品質は、形式的手法の分野[6)]から議論されてきた。近年、情報セキュリティに関する記述力を強化した形式モデル[7)]の研究も進められているが、オブジェクト指向プログラミング言語や UML など産業界の主流のモデルとのギャップも大きくなっている。従来の形式手法は実用上の面に大きな難点がある。

セキュアなソフトウェアは、あらゆるソフトウェア開発現場の課題である。産業界で広く利用可能な新しいセキュアソフトウェアの設計が必要といえる。我々は、ソフトウェアへの入出力モデルに着目し、セキュアソフトウェアな準形式的仕様書 $\mathcal{M}$ を定義する。本稿では、この簡単なモデルが、過去のセキュリティ脆弱性の事例やセキュアコーディングの経験をカバーできるか検討する。本仕様書 $\mathcal{M}$ は、プログラミング言語の型システムと等価であるため、原理的にコンパイラの型チェック機能によって機械的に検証できる。本稿では、プログラミング言語と品質保証の関係も議論する。

本稿は、次の通り構成される。第 2 節では、ソフトウェア設計とセキュリティに関して定義する。第 3 節では、準形式的なセキュアソフトウェア仕様書 $\mathcal{M}$ を定義する。第 4 節は、セキュアソフトウェア仕様書 $\mathcal{M}$ とプログラミング言語の関係を議論する。第 5 節は本稿をまとめる。

† 横浜国立大学大学院工学研究院
Graduate School of Engineering, Yokohama National University
†† 高エネルギー加速器研究機構
High Energy Accelerator Research Organization

☆ 本研究は、著者が工学院大学「セキュアシステム設計技術者育成講座」のゼミで議論した成果を反映している。

2. セキュリティ設計と限界

本節では、まずセキュアなソフトウェアを議論し、本稿における情報セキュリティと設計の関係を定義する。

2.1 情報アシュアランス

あるソフトウェアに脆弱性が発見されたとき、そのソフトウェアは「危険」と判断される。脆弱性の発見によって、セキュアなものがセキュアでなくなる。しかし、現実問題として、ある規模以上のソフトウェアには必ず脆弱性が含まれている。例え、既知の脆弱性に対し全て完全な対応を行っても、攻撃者は常に未知の脆弱性を発見する努力を続けている。

情報セキュリティは、定性的（もしくは定量的な）基準なく、感覚的な議論がされやすい。ここでは、感覚的なセキュリティとは区別して、情報アシュアランス (information assuarance) の立場から議論を進めたい。

定義 2.1 セキュアソフトウェアは、正常な動作が仕様化され、逆に振舞いは常に仕様書どおりである。

定義 2.1 に従えば、ソフトウェアの安全性は脆弱性の発見から独立する。もし脆弱性が存在しても、仕様書どおり動作すれば、そのソフトウェアはセキュアである。(問題は、仕様書にある。) プログラマは、個人としてのセキュアコーディングのスキルを求められず、仕様書にしたがってコーディングし、更にテストプランを作成・実行することでセキュアソフトウェアを開発することが可能になる。

2.2 アドバーサリ仕様書

次は、仕様書の種類をみていこう。従来のソフトウェア仕様書は、ソフトウェアの機能面 — 何が必要か (must)— に関心が集まっていた。そのため、ソフトウェア仕様書は機能仕様書とも呼ばれる。機能仕様書をどんなに厳密に記述してもセキュアな仕様を作成することは難しい。

一般に、安全性 (セキュリティ) を議論する場合は、どう機能すべきでないか (must not) が重要となる。これは、従来の機能仕様書とは別の視点といえる。本稿では、アドバーサリ仕様書 (adversary specification) と呼んで、機能仕様書と区別する。

定義 2.2 (アドバーサリ仕様書) セキュアソフトウェアの仕様書は、攻撃者を想定して、禁止される動作を定義しておく必要がある。

ソフトウェア仕様書 (機能仕様書) は、ソフトウェア開発中に変更されないことが望ましい。これに対し、アドバーサリ仕様書は、開発中のみならずリリース後に更新される可能性がある。ソフトウェア脆弱性に対するパッチは、アドバーサリ仕様書がカバーする領域である。ただし、ソフトウェア機能を開発することに比べ、既存の機能へのアクセスを制限することはそんなに大仕事ではない。

3. 仕様書 M の定義

本節では、準形式的なセキュアソフトウェア仕様書 M を定義する。本稿では、仕様書の記述を機械検証可能かどうかによって、形式的か、非形式的かを区別している。仕様書 $\mathcal{M}$ は、プログラミング言語の型チェック機構を使うことで、部分的に検証可能である。

3.1 入出力モデル

ソフトウェア仕様書は、ソフトウェア全体もしくはコンポーネント機能に関する記述である。最も簡単な機能の記述法は、データの入力 (input) に対する出力 (output) を振る舞いとして定義する方法である。これは、入出力モデルや signature-based model と呼ばれ、Web サービスの WSDL、CORBA/IDL, Java 言語/interface 宣言など、広く利用されている。

データの入出力は、プログラミング言語のプロシジャコール (procedure call) とみなすことができる。つまり、入出力の種類はプロシジャ名 (関数名、メソッド名)、入力は引数 (パラメータ)、出力は戻り値となる。プログラミング言語では、複数の引数値が認められるが、これを戻り値と同様にひとつであると仮定しても一般性は失わない。

入出力メッセージ m は、p をプロシジャ名、t, t' をそれぞれ入力、出力のタイプとすると、$p : t \mapsto t'$ となる。ソフトウェアは、入出力メッセージの集合として定義することができる。これを使えば、入出力モデルはソフトウェアの最小仕様書として定義できる。

定義 3.1 (M) ソフトウェアの最小仕様書 M は、入出力メッセージの有限集合 $M = \{m_1, m_2, .., m_n\}$ である。

もちろん、ソフトウェアはより複雑であり、単純な入出力メッセージだけでモデル化できない。仕様書 M は、プロシジャコールの順序 (ステート) や同期などの要素が省略されている。この省略がセキュア設計に与える影響に関しては、次節で議論する。

3.2 タ イ プ

次にメッセージのフォーマットを定義するタイプをみていこう。

タイプは、実現値/インスタンスを持つ。ここでは、オブジェクト指向言語であるなしに関わらず、単純にオブジェクトと呼ぶことにする。タイプとオブジェクトの関係は、次のインスタンス関数 **I** で定義できる。

- $T = \{t_1, t_2, .., t_i..\}$. タイプ (名) の集合

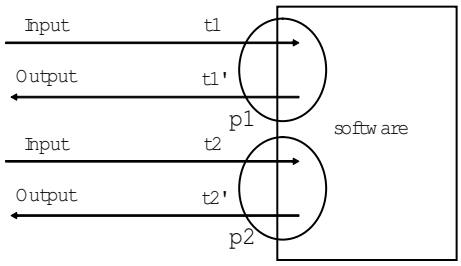

図 1 ソフトウェア仕様 $\mathcal{M}$

- $O = \{o_1, o_2, .., o_i..\}$. オブジェクトの集合
- $\mathbf{I} : T \mapsto 2^O$. インスタンス関数

タイプチェック (type checking) は、あるオブジェクト o がタイプ t の仕様を満たすかどうか判定することである。次のとおり定義できる。

定義 3.2 (タイプチェック) 与えられたオブジェクト o がタイプ t であるのは、$o \in \mathbf{I}(t)$ のときだけである。

以後、タイプチェックを使えば、タイプに属さないオブジェクトは全て除外できるものと仮定する。

3.3 セキュアなソフトウェア仕様

最後に、タイプを用いてセキュアなソフトウェア仕様 $\mathcal{M}$ を作成する。通常の仕様 M と区別するため、記号 $\mathcal{M}$ を用いる。

まず安全なタイプ (secure type) から定義しよう。あるプロシージャ($p : t \mapsto t'$) において、有害な入出力オブジェクトの集合 $\underline{O}$,$\underline{O'}$ がわかっていると想定する。☆このとき、$\mathbf{I}(t) \cap \underline{O} = \emptyset$ なら、t は安全なタイプである。同様に、$\mathbf{I}(t') \cap \underline{O'} = \emptyset$ なら、t' は安全なタイプである。

我々は、タイプ t がセキュアであるということ強調するため、τ と書くことにする。もし $p : \tau \mapsto \tau'$ なら、プロシージャp はセキュアである。これで安全なソフトウェア仕様が定義できる。

定義 3.3 セキュアなソフトウェア仕様 $\mathcal{M}$ は、$\mathcal{M}$ 上の全てのプロシージャがセキュアである。

4. 既知の脆弱性に関する評価

ここでは、ソフトウェア仕様書 $\mathcal{M}$ がどの程度、既知のソフトウェア脆弱性に対して有効か議論する。

4.1 既知の脆弱性

CERT は、ソフトウェア脆弱性に関する情報収集や勧告を行う機関である。CERT が提供するデータベースによると、報告されたソフトウェア脆弱性の大半は、buffer overflow(heap overflow も含む) に起因している。

☆ 有害な入力とは、例えば誤動作や破壊、特権奪取が可能なオブジェクトであり、有害な出力とは情報漏洩を招くオブジェクトである。

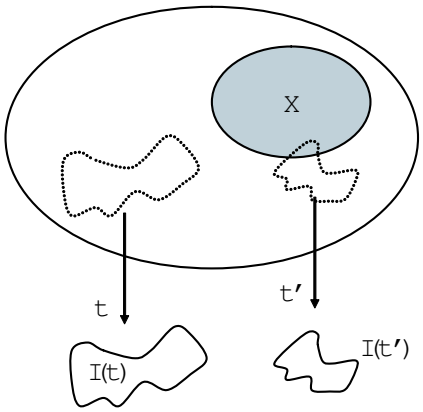

図 2 セキュアなタイプと危険なタイプ

表 1 代表的なソフトウェア脆弱性

Vulnerabilities	$\mathcal{M}$
Illegal UTF8（I18N) Characters	O
URL/File Path	O
Integer Coercion (Wrap-Around)	O
Cross-Site Scripting	O
SQL Injection	O
Pseudo-Random Number	x
Time of Check, Time of Use	x

Buffer overflow は、C/C++のプログラミング言語上の弱点を利用したものであり、データ入力に対しバッファ長をチェックするコードを追加することで回避できる。しかし、古いライブラリを中心に多くの脆弱性が内在しており、ソフトウェア開発者がコーディングの努力で回避するのは難しい。最近は、buffer overflow を抑制する機能 (stack guard) がついたコンパイラの利用が増えている。

Buffer overflow 以外にも知られた脆弱性が存在する。表 4.1 は、文献[1)~3)] からまとめたソフトウェア脆弱性の名称である。これらの脆弱性は必ずしも全てを網羅していないが、実際問題として設計者が開発者にこれ以外の脆弱性対策を期待することは難しい。

右列は、ソフトウェア仕様書 $\mathcal{M}$ によって入出力を正しくチェックすることで回避できるか示している。乱数生成の脆弱性 (Pseudo-Random Number) は、内部のアルゴリズムの問題なため、ソフトウェア仕様 $\mathcal{M}$ だけでは記述できない。また、ToCToU 攻撃は、マルチプロセス (マルチスレッド) 実行環境において同期処理の欠陥に起因するため、同様にソフトウェア仕様 $\mathcal{M}$ では記述できない。

4.2 未知の脆弱性

次に、あまり知られていない脆弱性、もしくは発見されていない脆弱性に関してみていこう。

最近、報告される新しい脆弱性のひとつに integer

overflow[8] がある。定義はいろいろ存在するが、integer 変数の reduced modulo の性質を利用して、heap overflow と組み合わせて攻撃に利用されることが多い。Integer overflow は、ソフトウェア実行による累積的な変数値 (の変化) から引き起こされるため、単純な入力チェックでは防ぐことはできない。個々の入力値は正しくても、Integer overflow は発生する。

同様に、アルゴリズムの最悪計算量を狙った攻撃の可能性も指摘されている。例えば、Crosby ら[9] は、ハッシュ表の最悪ケースを再現する順序で入力を入れることで、ソフトウェア機能を停止させる手法を報告している。このように、複数の入出力にまたがって検査する必要ある場合は、仕様書 $\mathcal{M}$ では記述できない。

ただし、現在問題となっている多くのソフトウェア脆弱性は、単純な入出力のチェックに基づくものである。仕様書 $\mathcal{M}$ は、実用的な立場から、十分に多くのソフトウェア脆弱性を除外できるといえる。

5. 機械検証性

ソフトウェア仕様書 $\mathcal{M}$ は、概念的にはプログラミング言語のタイプと同等である。本節では、プログラミング言語のタイプチェックを用いて、ソースコードが仕様書 $\mathcal{M}$ に準拠しているか機械的に検証できるか考察する。

5.1 タイプセーフ

まず、タイプセーフなプログラミング言語とセキュアソフトウェアの関係をまとめておこう。

タイプは、一般的にデータ構造 (data structure) とオペレータ (operations) から構成される。古典的プログラミング言語 (C 言語など) やデータベース、XML ではデータ構造が重視されるが、オブジェクト指向プログラミング言語ではオペレータが重視される。

今、タイプ t のある変数オブジェクト o を考える。タイプセーフとは、t で定められたオペレータによって操作した後の値 o' が常に $o' \in \mathbf{I}(t)$ である。例えば、C 言語はタイプセーフではない。例えば、`int` 型の場合、整数演算子以外にポインタ演算子によって自由に値を変更することができるからである。これに対し、Java 言語は、private 修飾子を用いることで、タイプセーフなクラスを作成することができる。

タイプセーフなプログラミング言語を用いれば、一度チェックした安全なタイプであると判定されたオブジェクトは実行中において常に安全であることが保証される。逆に、タイプセーフでない言語の場合は、コード境界においてタイプチェックするコードを意図的に追加する必要がある。(もちろん、これは非効率な方法である。)

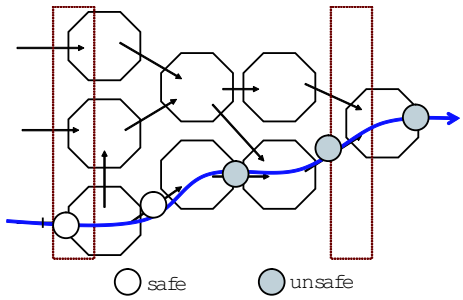

図 3 コード境界とタイプチェック

タイプセーフ言語のコンパイラによって検証されたソフトウェア仕様書 $\mathcal{M}$ は、仕様書通りに有害なオブジェクトの入出力を防ぐことが保証される。

5.2 タイプと表現力

タイプセーフは、開発されたソフトウェアが仕様書 $\mathcal{M}$ のとおり動作することを保証する最初のテクニックとなる。C/C++言語は、残念ながらタイプセーフでないため、C/C++言語で仕様 $\mathcal{M}$ を記述してもそのとおりに動作するとは限らない。ただし、タイプセーフ言語で仕様書 $\mathcal{M}$ を記述すれば、そのままセキュアソフトウェアになるとは限らない。一番の問題は、プログラミング言語がサポートするタイプが十分な表現力を備えていない場合がある。そのような場合、他のよく似たタイプで代用することが多く問題が発生する。

例えば、月 (month) を表現するとき、`int` 型で代用する場合がある。次の代入は、明らかに間違っているが、タイプチェック機構では検出することができない。

```
int month = 13
```

タイプ代用の問題は、タイプセーフの効果がなくなる点である。つまり、プログラマは、タイプセーフでない言語と同じように、コード境界では `ismonth(int month)` のようなチェックコードを追加しなければならない。

また、文字列はソフトウェア脆弱性の原因となりやすいタイプである。文字列は、極めて表現力が豊かであり、何らかの解釈器によって副作用が発生する可能性がある。

次は、SQL Injection の例である。変数 name を String で代用しているため、任意の SQL 文が埋め込み可能となる。その結果、データベースが破壊されたり、データが盗まれたりする。

```
String name;
String sql = "select * from name_tbl "
```

```
+ "where name='" + name + "'"
```

文字列をチェックする手法として、正規表現 (regular expression) が一般的に利用されているが、広く普及したプログラミング言語でタイプとして正規表現をサポートしていない。

タイプセーフなオブジェクト指向言語の場合、タイプセーフなクラスを定義し、タイプ代用を避けることで、仕様書 $\mathcal{M}$ をプログラミング言語上で記述し、そのセキュアソフトウェアとして動作することが保証される。

6. 結論と展望

ソフトウェア脆弱性のないソフトウェアを開発することはほぼ不可能である。ソフトウェアの設計段階からセキュリティ対策を行い、品質管理をすることが求められている。

本研究では、プログラミング言語のタイプチェック機能を用いて、ソフトウェアのセキュアさを保証する方法に関して考察を進めた。ソフトウェア仕様書 $\mathcal{M}$ は、ソフトウェア開発の現場で対応可能なセキュリティ対策を十分に記述でき、更にタイプセーフな言語で記述することで、設計書どおりに動作が保証できた。

今後は、より複雑な脆弱性への記述力との強化とプログラミング言語との融合を進め、より生産性高くセキュアソフトウェアの開発を支援できるツールを議論していきたいと考えている。

謝辞 平成 16 年 11 月に急逝された塚本克治教授 (工学院大学工学部情報工学科) に捧げる。また、有益な議論に協力していただいた工学院大学 CPD センター第 1 期受講生へ謝意を表したい。

参 考 文 献

1) John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley. September, 2001.
2) John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'REILY, July, 2003.
3) Michael Howard and David LeBlanc. *Writing Secure Code (Second Edition)*. Microsoft Press, 2003.
4) Anup K. Ghosh, Chuck Howell, and James A. Whittaker. Building Software Securely from the Ground Up *IEEE Software*, pp. 14-16, vol.19, No. 1, 2002.
5) Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System *IEEE Software*, pp. 18-25, vol.19, No. 1, 2002.
6) Howard Bowman and Johon Derrick. *Formal Methods for Distributed Processing* Cambridge University Press, 2001.
7) Abadi, M. and Gordon, A. D. A Calculus for Cryptographic Protocols: The spi Calculus, *Information and Computation* Vol. 148, No. 1, pp. 1-70, 1999.
8) Dave Ahmad. The Rising Threat of Vulnerabilities Due to Integer Errors, *IEEE Security and Privacy*,pp. 77-82, No.4, Vol.1, 2002.
9) Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks, In *Proc. of Usenix Security 2003*, August 2003.