アジャイル Web アプリケーション開発における ソフトウェアセキュリティ保証のための 反復型評価手法に関する研究

宗藤 誠治

博士 (情報学)

総合研究大学院大学
複合科学研究科
情報学専攻

平成 27 年度 (2015)

2016 年 3 月

本論文は総合研究大学院大学複合科学研究科情報学専攻に
博士 (情報学) 授与の要件として提出した博士論文である.

審査委員:

(主査)	吉岡信和	国立情報学研究所 / 総合研究大学院大学
	石川 冬樹	国立情報学研究所
	胡 振江	国立情報学研究所 / 総合研究大学院大学
	中島 震	国立情報学研究所 / 総合研究大学院大学
	鷲崎 弘宜	早稲田大学/国立情報学研究所

(主査以外はアルファベット順)

Towards Iterative and Incremental Evaluation for Software Security Assurance of Agile Web application Development

Seiji Munetoh

Doctor of Philosophy

Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies (SOKENDAI)

March, 2016

A dissertation submitted to the Department of Informatics, School of Multidisciplinary Sciences, The Graduate University for Advanced Studies (SOKENDAI) in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Advisory Committee:

Nobukazu Yoshioka (Chair)	National Institute of Informatics / The Graduate University for Advanced Studies (SOKENDAI)
Fuyuki Ishikawa	National Institute of Informatics
HU Zhenjiang	National Institute of Informatics / The Graduate University for Advanced Studies (SOKENDAI)
Shin Nakajima	National Institute of Informatics / The Graduate University for Advanced Studies (SOKENDAI)
Hironori Washizaki	Waseda University / National Institute of Informatics

(Alphabetic order of last name except chair)

論文要旨

ウォーターフォール型からアジャイル型へとソフトウェア開発プロセスが変化する中、ソフトウェアのセキュリティ保証の新しい仕組みが必要とされている。ウォーターフォール型と呼ばれる従来の開発プロセスでは、開発の各段階（要求、設計、実装、テスト）に対応したセキュリティ保証に、セキュリティ専門家を含む多くのリソースを費やすことで、脆弱性のないソフトウェアの実現を目指してきた。こうした取り組みは、特に大規模なソフトウェアやシステムの新規開発に広く実践されている。一方、アジャイル型と呼ばれるソフトウェア開発では、小規模な開発チームが短いリリースサイクルで反復的に開発を進める。そのため、脅威分析や網羅的なセキュリティテストのように手間と時間のかかる作業を頻繁に実施することは難しい。また、セキュリティに関する専門家の不在、開発者が十分にセキュリティに関する知識を持っていない事も課題として挙げられる。

本研究では、反復的な開発プロセスにセキュリティ保証を効率的に組み込む方法として、アジャイルソフトウェア開発手法の一つであるテスト駆動開発を拡張することで、開発者自身によるセキュリティ要求の把握とセキュリティテストの記述及び実行を実現する。その際の課題である、設計及びコード実装に依存する脆弱性への対応と、セキュリティテストで必要とされる網羅性の保証を実現するために、知識と作業効率の両方の観点から開発者を補佐する新しい手法を提案する。

提案手法では、アプリケーションの実装コードの中に現れるコマンドについて、そのセキュリティに関する情報を抽象化、集約、知識化する。具体的には、検査対象のアプリケーションを、アジャイル開発の中での実装部分（アプリケーションコード）と、フレームワークのようにアプリケーションが利用している部分 (フレームワークコード) の２つに分離し、後者の機能を抽象化した「コマンド抽象化ライブラリ」を作成する。ここで言うコマンドとは、アプリケーションの実装コードが呼び出すアプリケーションフレームワークの提供する様々な機能を示す。次に、コマンド抽象化ライブラリを用いて、実装コードの静的解析から、アプリケーションの振る舞いモデル生成、セキュリティ要求の抽出、(静的な) セキュリティ解析、(動的な) セキュリティテストカバレッジの計測をツール化することで、開発者のセキュリティ保証作業を補佐する。

ここで作成するライブラリにはフレームワークが提供するすべてのコマンドを登録する。その中でアプリケーションの振る舞いと、セキュリティに関係するコマンドの特性を分類することでフレームワークの提供するセキュリティ機能を抽象

化する。セキュリティに関係するコマンドを、Security Command, SC：セキュリティ機能を実現するコマンドと、Risky Command, RC：その使用がセキュリティ上のリスクとなる可能性のあるコマンドの 2 種類に分類する。SC に分類されるコマンドは、例えばアクセス制御に関するコマンドであり、これらは主にアプリケーションのセキュリティ要求及び設計（デザインの脆弱性）に関係する。RC に分類されるコマンドとしては、インジェクション攻撃の対象となるコマンドであり、主にコード実装 (実装の脆弱性) に関係する。以上のように、コマンドを 2 つに分類することで、提案手法は設計と実装の双方のセキュリティの問題に対応する。

設計に関する脆弱性に対応するためには、アプリケーションの動的な振る舞いを把握する必要がある。そのため、その振る舞いに関するコマンドもライブラリに登録する。この情報を元に、アプリケーションコードの静的検証からセキュリティ検証モデルを効率的に生成する。モデルは制御フローモデルとデータフローモデルから構成され、アプリケーションのセキュリティ機能の検証には制御フローモデルを、インジェクション攻撃などの攻撃可能性の検証にはデータフローモデルを利用する。開発者は、ツール化された本手法を用いて、実装したアプリケーションのセキュリティ機能や問題箇所を迅速に把握し、セキュリティテストケースを作成する。

セキュリティテストのカバレッジについては、コード中に存在する SC と RC に対応するテストの有無から、テストカバレッジを計測する。開発者は SC については、セキュリティ機能の確認 (サンプリング)、RC については、その存在が脆弱性に繋がっていないことの確認のテストケースを作成する。これらは、カバレッジ情報を元に効率的にテストを配置する。

最後に、SC と RC にソフトウェアに関する体系的なセキュリティ知識を対応付ける。これにより、開発者がセキュリティ要求、脆弱性およびその対策方法、及びそれらについてのテスト手法に関する知識に、実装コード視点から参照することが出来るようになり、セキュリティ知識が不十分な開発者をツールがサポートする。

以上のように、提案手法では、セキュリティの観点から開発者が柔軟かつ迅速に要求、実装、テストの関係を把握することで、アジャイルソフトウェア開発に適合したセキュリティ保証を実現する。本研究で提案する実装コードレベルでのセキュリティ知識のライブラリ化は、アプリケーションのセキュリティ保証の新しい手法であり、反復開発及び開発者間でのセキュリティ知識の共有と利用を、ライブラリを介して実現する。

提案手法の評価にあたり、アジャイルソフトウェア開発との整合性の確認の観点から次の 3 つの目標を設定した。1）要求や設計に起因するセキュリティの問題と、実装に起因するセキュリティの問題とを統一した手法で取り扱えること (ツール化)、2）コマンド抽象化ライブラリの作成と保守が開発者にとって大きな負担とならないこと、3）アプリケーション付随の回帰テストでセキュリティ保証を行

える（テスト駆動開発にセキュリティテストが組み込める）ことである。提案手法を用いることによってこれらの目標が実現できることを、アジャイル型開発を代表する Web アプリケーションフレームワーク、Ruby on Rails 用のセキュリティツール（RailroadMap）の開発を通して確認した。

Abstract

In tandem with the migration of the software development process from waterfall to agile, there is a need for novel methods for security assurance of software. In the traditional development process called waterfall, it has been aiming to achieve security assurance that with no vulnerable software by spending a lot of resources, including security experts, for each stage of development (request, design, implementation, test). These efforts are widely practiced especially in the new development of large software and systems. On the other hand, in the software development, called agile, small development team proceeds the iterative development in short release cycle. Therefore, it is hard to frequently implement labor and time-consuming works such as threat analysis and comprehensive security test. In addition, the absence of security experts and the insufficient security knowledge of developers are a common problem.

In this study, we were intended to support elicitation of security requirements and carrying out security test by developers themselves as a way to incorporate a security assurance to the iterative development process efficiently. To achieve this we extend the test-driven development, which is part of the agile software development methodologies, The main challenges are handling of vulnerability from both the design and the code implementation and completeness of the security test to ensure the security assurance. Therefore, we propose a novel method to assist the work of developers from the point of view of security knowledge and work efficiency.

This study realizes the automation of security assurance by abstracting security related information at the command level. The term “command” indicates various functions performed by the application framework which called from application implementation code. First, the target application is classified into two parts, implementation code in Agile development (application code) and framework side code called by application code (framework code). The framework code is abstracted as “command abstraction library”. Then, a tool generates a security verification model from application code, extracts the security requirements, analyses a code security statistically, and measures a security testing coverage by using the library.

All of the commands are registered into the library, and classify the char-

acteristics of the commands related to the behavior of the application and security. A command related to security is classified into two types, SC (Security Command) and RC (risky command). The commands, which are classified as SC, are mainly performed a security function related to the security requirements and design of the application (design vulnerability), for example an access control. The commands, which are classified as RC, are a risky command that related to potential vulnerability code (implementation vulnerability), for example various types of injection attack. As mentioned above, this classification supports security issues corresponding from both design and implementation of application.

In order to deal with the vulnerability relates to the application design, it is necessary to understand the dynamic behavior of the application. For this reason, a command relevant to the application behavior is also registered in the library. Consequently, a tool effectively generating a security verification model from static analysis of the application code based on this information The model consists of a control flow model and a data flow model. The control flow model is utilized to validate the security features of application. The data flow is utilized to verify the possibility of attacks, such as injection attacks. Developers quickly understand the security functions and weaknesses, and create a security test case of them by using a tool which support proposed method.

To assess the security test coverage, the analysis tool counts the presence or absence of the test case corresponding to the SC and RC embedded in the code. Developers create a test case to verify the behavior of security functions (sampling), and the use of the RC does not the cause of vulnerability. They can efficiently arrange test cases based on this coverage information.

Finally, systematic knowledge of the software security is associated with the SC and RC in the library. As a result, the developers will be able to refer the security requirement, vulnerability and countermeasure, and testing method of them from the implementation code point of view. The tool supports developer who has insufficient security knowledge.

As described above, the developer flexible and quickly understands of the relationship between the requirements, the implementation and the test from the view point of security by the proposed method. This conforms the security assurance to agile software development. Utilizing a library of security knowledge, which linked with command at the implementation code, for security assurance is a new method to share and use of security knowledge between iterative development cycle and application developers.

The following three goals are set to evaluate the proposed method from the point of view of consistency of the Agile software development: 1) The method can handle a security problem due to both the design and implementation in a unified approach (or tool), 2) A maintenance of the command abstraction library is not a big burden, 3) Regression testing by developers supports security also (security test is incorporated into test-driven development). The evaluation was executed through the development of security tool (RailroadMap) for the Ruby on Rails Web application framework that represents the agile development.

目 次

図 目 次

表 目 次

第1章　序論

情報技術は社会インフラを構成する重要な要素であり、情報セキュリティの問題は個人や組織、社会に様々な損害を与えるリスクの一つとして認識されている。この問題の主な原因は、情報システムを構成するソフトウェアに含まれる「脆弱性」である。脆弱性とは、攻撃者に悪用されるソフトウェアのバグであり、脆弱性の無い「セキュアなソフトウェア」の実現が社会的な要請であるといえる。これまで、様々なセキュリティ保証の手法や開発プロセスの研究の結果として、セキュリティ対応を組み込んだソフトウェア開発の実践が普及している。特に、大規模かつ計画重視のウォーターフォール型のソフトウェア開発においては、セキュリティへの取り組みの整備と成熟化が進んでいる [50]。しかしながら、ソフトウェアを実現するプログラミング言語、開発手法、攻撃手法は絶えず変化しており、セキュリティ対応が後手に回っているのが現状である。

一例として、近年普及が進んでいるアジャイルソフトウェア開発手法 [18] は、従来の計画重視の開発プロセスに則したセキュリティ保証の手法との不整合が指摘されている [19][15]。例えば、セキュリティ要求の定義（文書化）はセキュリティ評価の礎であるが、アジャイルソフトウェア開発のように要求が絶えず変化し、またその変化の速度が早い開発では、文書化に依存するセキュリティ保証手法は適用が難しい。また、従来のウォーターフォール型のソフトウェア開発では、セキュリティ保証は第三者によるセキュリティレビュー、テストなどの品質保証プロセスで実現されているが、ジャイルソフトウェア開発では、小規模なチームが密に連携して、短い反復（イテレーション）サイクルで開発をすすめるため、第三者の参画によるセキュリティ保証手法の適用は難しい。そこで、テスト駆動開発 [17]、反復型開発などのアジャイルソフトウェア開発手法の長所を損なうことなく実施可能なセキュリティ保証の手法を確立し、実際に現実のアジャイルソフトウェア開発環境に適用することが本論文の目的である。

最初に本論文で扱うソフトウェアの種類とセキュリティ問題について整理しておく。

本論文では Web アプリケーションフレームワークを用いて開発される Web アプリケーションを対象としている。この選択の理由は次の 2 点からなる。1）インターネットに接続して稼働する Web アプリケーションは攻撃対象となりやすく、実際に様々なセキュリティ上の問題（脆弱性）が発生し社会問題となっているアプリケーションの開発分野であること。2）アジャイルソフトウェア開発が広く普

及し実践されているアプリケーションの開発分野であること。

ソフトウェア開発には様々なセキュリティの問題が存在するが、本論文で取り扱うセキュリティの問題は、アジャイルソフトウェア開発を実践しているアプリケーション開発者が対応すべきであるセキュリティの問題である。これには設計と実装の 2 つの視点がある。設計に関しては、セキュリティ要求にしたがって、正しくセキュリティ機能が実装されていること、実装に関しては、実装されたコードにおいて脆弱性が存在しないことである。アジャイルソフトウェア開発の場合、ソフトウェア開発者は実装（コーディング）のみならず、アプリケーションの機能や設計にも関与する。本論文の提案手法は、そうした Web アプリケーションの開発者による実施を想定したセキュリティ保証の手法である。

本章ではまず 1.1 節で本研究の背景と目的を述べ、1.2 節で 具体的な動機例を示したあとに、1.3 節で提案手法の要約を述べるとともに、本論文全体の構成を示す。

1.1 研究の背景と目的

本研究は、アジャイルなソフトウェア開発手法に適用可能なセキュリティ保証の実現を目的としている。まず、従来のソフトウェアセキュリティ保証 (Software security assurance) の仕組みとアジャイルソフトウェア開発手法の課題の概略を示す。

現在、様々なソフトウェア開発において、セキュリティの問題に対応したソフトウェア開発プロセスの実施が必要とされている。ソフトウェア開発のセキュリティ成熟度を示すメトリックスとして、OpenSAMM[1] や BSIMM[2] などのメトリックスが提案されている。こうしたメトリックスは、開発されるソフトウェアの安全性を高める組織的で計画的な取り組みの計測に利用されており、計画重視のウォーターフォール型のソフトウェア開発に適合する形で整理されている。

一方、近年の Web アプリケーション開発などでは、より適応的なアジャイルソフトウェア開発手法が用いられるようになってきている。この技術的な背景には、計算機の性能向上、スクリプト言語を用いた動的なアプリケーション開発、アプリケーションフレームワークの成熟、ソフトウェアのオープンソース化とインターネットを介したオープンな共同開発、そして、クラウドを活用した柔軟なアプリケーションの開発及び実行環境の普及などのソフトウェア開発環境の変化により、アプリケーションの実装コード量や開発工数が大幅に削減されたことがあげられる。しかしながら、外部のセキュリティ専門家によるレビューやテストは、小規模なチームによる密な連携で短い周期でのリリースを繰り返す開発形態では実施が難しい。また、開発者が十分にセキュリティに関する知識を持っていないため

[1] http://www.opensamm.org/
[2] http://bsimm.com/

に、十分なセキュリティ要求の把握やセキュリティテストの実施がソフトウェアのリリース前に実施できていないことも課題となっている。アジャイルソフトウェア開発では、要求と実装の双方が高い頻度で変化するため、脅威分析やセキュリティ要求定義などの文書化作業とその保守のように工数のかかる作業は、開発者にとって大きな負担となる。

この問題に対する一つの解決方法は、セキュリティの問題をできるだけアプリケーション開発者が意識しなくてもよい仕組みにすることである。プログラミング言語や Web アプリケーションフレームワークが、セキュリティ機能を標準で提供することで、開発者によるセキュリティ対策の負担は低減することができる。実際に、多くの Web アプリケーションフレームワークが、フレームワーク側でのセキュリティ対応をバージョンの更新とともに進めている。しかしながら、全てのセキュリティの問題に対して、フレームワークやプログラミング言語側で対応することはできない。セキュリティの問題は要求定義、設計、実装など様々な開発段階で発生するが、フレームワーク側で解決できるのは主に (Web の場合はクロスサイトスクリプティングや SQL インジェクションなどの) アプリケーションドメイン固有の実装の問題である。一方で、こうしたフレームワークが提供するセキュリティ機能を、何らかの実装上の理由で無効化する場合には、開発者による安全性の確認が必要である。また、新しいセキュリティ上の問題（脆弱性）が見つかった場合も、その問題への対処はフレームワーク側で対応ができるまでは開発者の責任となる。

別の解決方法は、セキュリティ保証手段のツール化（自動化および軽量化）による、開発者のセキュリティ保証にかける負担の低減である。例えば、計量な静的検証ツールによるソースコードレベルでの静的なセキュリティテストはアジャイルソフトウェア開発でも広く活用可能である。このような静的検証ツールは、コード上で問題箇所を指摘できるため、問題の修正が容易であり、開発者にも比較的扱いやすい。ただし、この種のツールの検査能力や精度（False-Positive、 擬陽性）と実行時間にはトレードオフがあり、対応できる脆弱性の種類にも限りがある。一方アプリケーションを動作させて検証する脆弱性スキャナによるセキュリティテストは、ツールの実行時間が長いこと（数時間から数日)、指摘された問題の修正にはツールの実行結果を解析する必要があること、ツールの利用に熟練が必要なことが原因で、開発者がコード実装と共に用いるには負担が大きい。

このように、既存の手法やツールをアジャイル開発の中で用いるには様々な課題があるが、そうしたアジャイルソフトウェア開発との不整合が指摘されたセキュリティ保証の仕組みを、できるだけ軽量な形で自動化し、開発フレームワークに組み込むことができるのであるならば、開発者自身によって、コード実装作業と平行して、より広範囲のセキュリティ保証作業を実施することが可能となり、アジャイルソフトウェア開発のセキュリティ品質が向上するはずである。したがって、本研究では、セキュリティ要求やテストと実装との関係を、アプリケーションの実

装コードから自動的に抽出し、一貫性を確認することで、セキュリティ保証をアプリケーションの開発工程に自然に組み込む手法の実現を目的とする。

1.2 動機となったWebアプリケーションのセキュリティ機能の実装例

本節では、研究の動機であるアジャイルなWebアプリケーション開発におけるセキュリティテストの課題について、具体的な実装の例を参照して説明する。

Webアプリケーションフレームワークの一つであるRuby on Rails [3]は、アプリケーションを実装する際のコードの少なさが特徴の一つであり、アプリケーションの記述で用いるスクリプト言語のRubyさえ習得していれば、誰でも簡単にWebアプリケーションを構築することができる。しかしながら、実際のWebアプリケーションを開発する際には、開発者は適切なセキュリティ機能をアプリケーションに組み込む必要がある。Webアプリケーションに必要となるセキュリティ機能にはアクセス制御や、アカウント及びセッションの管理、扱うデータの暗号化と鍵管理、通信の暗号化等がある。Railsのフレームワーク自身でも、簡単な認証機能をサポートしており、実用レベルのより高機能なアカウント管理や、Role-based Access Control (RBAC)をサポートしたい場合には、アプリケーション独自にそうした機能を実装するか、そうした機能を提供する外部パッケージを利用することも可能である。広く普及している外部パッケージの利用は、Webアプリケーション開発速度を加速するとともに、個別のアプリケーションで独自にセキュリティ機能を開発するよりも、一般に高品質である。一方、開発者が外部パッケージが提供するセキュリティ機能を十分に理解せずに利用した場合には、利用方法のミスや、設定のミスがアプリケーションの深刻な脆弱性となる危険性もある。

この節では、Railsにおけるアクセス制御機能の実装を例に、コード実装でのミスがどのようにセキュリティ上の問題（脆弱性）となるかを示す。RailsはMVC(Model-View-Contoller)アーキテクチャを採用しており、クライアント(Webブラウザ)からのリクエストは、フレームワークによりルーティングされ、対応するController [4]で処理される。図1.1の例では、Railsを用いたアプリケーションで広く利用されている、認証の外部パッケージ、Devise[5]と認可の外部パッケージCanCan[6]を用いたコードを示す（Railsのアーキテクチャについては2.5節で詳しく説明する）。

Deviseが提供する認証機能は、Railsのフィルターコマンドとしてクラスの最初に配置することで、そのクラスのすべてのメソッドの実行前に呼びだされる。この例では、Userに関する情報のアクセスには認証が必要なため、Controllerクラスの UserControllerの最初に “before_filter :authenticate_user!” コマンドを配置することで、クラス内のすべてのメソッドでDeviseが提供する認証機能を利用する。開発者がこのコマンドの配置を忘れた場合、User情報に対する不正なアクセ

[3] http://rubyonrails.org/

[4] コード上ではControllerクラスの各メソッドがWebサイトの個別のURL(Uniform Resource Locator)のパス名に対応する

[5] https://github.com/plataformatec/devise

[6] https://github.com/ryanb/cancan

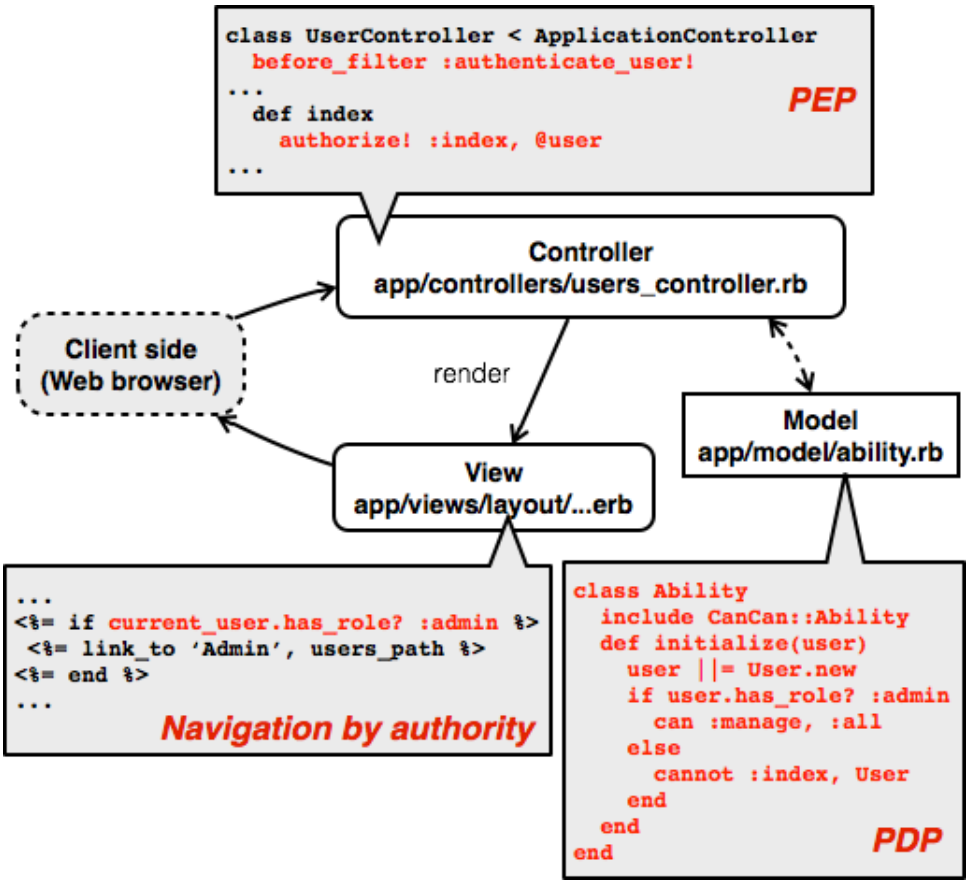

図 1.1: Ruby on Rails におけるアクセス制御実装例

スが可能となる。CanCan はアクセス制御実装で一般的な、Policy Enforcement Point (PEP) と Policy Decision Point (PDP) を分離した形式となる。PEP としては、クラス内の各メソッドに “authorize!” コマンドを配置し、メソッド実行前に権限のチェックを行う。CanCan の PDP は、 Model の実装コードの形でアクセス制御ポリシーを記述する。開発者はアクセス制御が必要なメソッドに PEP を正しく配置し、アクセス制御ポリシーを記述する。この例では、管理者権限 (admin) は Controller のすべてのメソッドにアクセスできるが、それ以外は User の一覧 (index) にはアクセス出来ない。したがって、PEP の配置ミス及び、ポリシーの記述ミスは、アクセス制御に関する脆弱性となる。以上が、アプリケーションの入力側での認証認可のチェックである。

一方、レスポンスとなる Web ページを生成するのが View であり、Rails では ERB[7] を用いて HTML を動的に生成する。この例のようにアプリケーションが複数のロールをサポートし、複数のロールで Contoller や View コードを共有する場合、利用しているユーザーの権限に応じてテンプレートで生成される Web ページの内容を制御する必要がある。この例では、管理者権限をもつ利用者だけが、

[7]Embedded Ruby によるページ生成のテンプレート

“users_path” 変数が示す URL(UserController の Index メソッド) へのアクセスが出来るため、“current_user.has_role? :admin” で利用者のロールをチェックし、生成するページの内容を選択している。View における権限チェックのミス自体は、Controller 側での権限のチェックが正しく実施されていれば、脆弱性にはならないが、利用者には、通常の利用の中で、不要なエラーメッセージを提示することになる。一般に、この種のエラーはナビゲーションエラーと呼ばれ、アプリケーションのバグとして修正する必要がある。

このように、Rails でアクセス制御機能をサポートする場合は、全体として整合した振る舞いになるように、開発者は MVC を構成するソースコードの様々な箇所に、セキュリティに関するコマンドを分散して配置する必要がある。その際に追加するコードは、それぞれ一行程度であるが、その実装ミスはアプリケーションのアクセス制御の重大な欠陥につながる。

以上のような、セキュリティ機能が正しく実装されていることを確認する手法としては、ソースコードのレビューと機能テストが一般的である。Rails の開発では、RSpec[8] や Cucumber[9] などが提供するテスト環境を利用することで、簡単にテスト駆動型の開発が実践できる。特に、Cucumber のような Behavior-driven Development (BDD) をサポートするテストフレームワークを利用する場合には、Gherkin と呼ばれる Domain specific language (DSL) を用いることで、UAT(User Acceptance Test) レベルの振る舞いの記述を用いた実行可能なテストケースの作成が可能である。この場合、リスト 1.1、 1.2 に示す例のように、自然言語に近い表現でテストシナリオを記述することが可能である。Web アプリケーションでは、クライアントは任意のページを指定して、アクセスを要求する事ができる [10] が、その際に、適切な権限を持たないクライアントのからのアクセスはエラーとして処理される。リスト 1.1 では 、「user として認証されてないクライアントが users ページにアクセスした場合に認証が必要な旨を警告する」というアプリケーションの振る舞いをテストしている。リスト 1.2 では、「 一般 user として Sign In しているクライアントが、管理者権限が必要な users ページにアクセスした場合に、権限が必要な旨を警告」という振る舞いをテストしている。この例で示すように、UAT レベルの DSL を用いたテスト記述は開発者以外のステークホルダーにもわかりやすく (実質的な) 仕様記述としても広く利用されている。

[8] http://rspec.info/, https://www.relishapp.com/rspec
[9] http://cukes.info/
[10] 強制ブラウジング (forced browsing) と呼ばれる

```
@attack
Feature: Attack by Anonymous user
As an anonymous user of the website, I want to access protected resources
...
  Scenario: I access to /users (id=C_user#index)
    Given I do not exist as a user
    When I access to "/users"
    Then I should see "You need to sign in or sign up before continuing." message
...
```

Listing 1.1: Cucumber による認証テスト

```
@attack
Feature: Attack by User
As a registered user of the website, I want to access protected resources

  Scenario: I sign in and access to /users (id=C_user#index)
    Given I am logged in
    When I access to "/users"
    Then I should see "Not authorized as an administrator." message
```

Listing 1.2: Cucumber による認可テスト

こうした記述は、セキュリティ機能の振る舞いの主要な一部分をテストするには適しているが、セキュリティテストで重要となる網羅性を、このテストフレームワークの中で実現することは現実的ではない。なぜなら、単純に網羅性を求めると、脆弱性スキャナのように、大量の組み合わせテストが必要となり、そうしたセキュリティに関する詳細なテストの記述は単純に開発者の負担を増やす。さらに、頻繁なアプリケーションの変更に伴って、アプリケーションに付随させた大量のセキュリティテストケースの更新実施が必要となり、開発のアジリティを大きく損なう。それに対して、もしセキュリティテストカバレッジも保証されたセキュリティテストケースを適切な数で開発者が作成し保守できるのであれば、それはアジャイルソフトウェア開発においても受け入れ可能である。

アジャイルソフトウェア開発におけるセキュリティ対策の難しさが指摘されるが、これは、従来の上流工程から始まる一連のセキュリティ保証の手法の適用が難しいことが主因のひとつにあげられる [19]。逆に、以下のような取り組みがフレームワークやツールの形で開発者に対してサポートすることができれば、アジャイルソフトウェア開発においても、十分なセキュリティ保証が実現できる可能性がある。

- 適切なセキュリティ要件の定義
- 適切なセキュリティ機能の選択
- 適切なセキュリティ機能の配置（実装、ポリシー記述）
- 適切なセキュリティ機能に関する設定の実施
- UAT によるセキュリティ機能の確認（ペネトレーションテスト）

- 必要十分なセキュリティテストケースの選択（テストケースの最小化）
- 要求と実装の変更に対する柔軟な対応

セキュリティ保証を実施する場合には、何が正しいアプリケーションの振る舞いなのかを、セキュリティテストのオラクルとして正しく定義する必要がある。クロスサイトスクリプティング (XSS) などの Web アプリケーション固有の実装に起因する典型的な脆弱性については、そのテストオラクルは自明である。しかしながら、アクセス制御や暗号化などのアプリケーションのセキュリティ設計と実装に係る脆弱性ついては、その検証には、アプリケーションのあるべき姿を定義した要件定義書や設計書がテストオラクルとして必要となる。一般的なアジャイルソフトウェア開発同様に、Rails の Web アプリケーション開発はコード（テストコード、ソースコード）開発中心に進むため、開発者の意図が要求として文書化され、最新の実装を反映するように管理できているとは言いがたい。

そうした場合に、文書化されていない、暗黙のセキュリティ要求にしたがって、セキュリティ機能が実装コード上（もしくはテストケース）に反映されると仮定することで、実装コードの解析から（暗黙の）セキュリティ要求を導き出して、その結果をレビューすることは可能である。実装コード上にセキュリティ機能が無い場合には、それがそもそも不要なのか、配置ミスなのかを判断して記録することで、必要最低限のセキュリティ要求の文書化が可能になる。

次に、セキュリティ・テストケースをテスト駆動開発に組み込む際には、Rails のフレームワークや、利用している外部セキュリティ機能パッケージが提供する振る舞い (攻撃に対する反応) を、開発者が理解している必要がある。また、作成したセキュリティ・テストケースのテストカバレッジが何らかの方法で定量化できるのであれば、適切な数のテストケースを作成し保守することが可能となる。

アジャイルソフトウェア開発では機能要求やその実装方法は絶えず変化している。そうした変化で生じるセキュリティ要求と実装、テストの不一致をいつでも検証 (逐次評価) 可能な状態にすることで、アジャイルソフトウェア開発プロセスと親和性の高いセキュリティ評価手法が実現できるはずである。

1.3 提案手法と貢献

アジャイルソフトウェア開発手法にセキュリティ保証の仕組みを組み込む上で課題となるのが、正しいセキュリティ要求の定義と、実装がセキュリティ要求に基づいて正しく実装されていることへの確認である。

本研究では、コマンドレベルでのソフトウェアの抽象化を用いて、先に挙げた課題に対応する。具体的には「コマンド抽象化ライブラリ」の形でアプリケーションフレームワークが提供する セキュリティに関係する情報をまとめる。ここで言う「コマンド」とはアプリケーションフレームワークが提供する API(Application Programing Interface) を指す。ライブラリでは、フレームワークが提供する個別のコマンド (API) の振る舞いとそのセキュリティ属性を抽象化するが、コマンド抽象化ライブラリを用いることで、セキュリティ保証の観点から、アプリケーション全体の振る舞いを抽象化することが可能となる。

コマンド抽象化ライブラリの対象は、検査対象のアプリケーションが利用しているフレームワークやライブラリである。実際の開発の対象となるアプリケーションコードは静的検証を用いて検査するが、フレームワーク部分についてはこのライブラリの定義を参照し処理する。つまり、アプリケーション・フレームワークの開発者全体で、フレームワークに関するセキュリティ知識をライブラリの形で整備、共有、保守する。結果として、この知識を活用することで様々なセキュリティ保証の仕組みが自動化できるようになる。

一般に、スクリプト言語を用いた Web アプリケーションフレームワークは、同一言語でフレームワークとアプリケーションがシームレスに記述されるため、すべてを検査対象とすると莫大なコード量が対象となり現実的ではない。また動的言語の静的検証は難しいため、検証精度が得られない。例えば、Rails のコード量は 10 万行以上だが、Ruby の静的解析ツール Laser [31] ではうまく処理できない。これはコード量の他に Ruby の動的な振る舞いの静的解析が難しいことも原因である。

一般に、フレームワーク部分はアプリケーション開発でのセキュリティ保証作業の対象外であり、フレームワーク部分はアプリケーションの反復開発と比べるとその変化の速度は遅い。そのため、利用しているフレームワークのソースコードを検査する必要はなく、利用しているフレームワーク部分の脆弱性の有無の確認が一般的である。しかしながら、アプリケーションの詳細な振る舞いはフレームワークの提供する機能で決まるため、正確にアプリケーションの実装コードを静的検証するためには、フレームワーク側のコードも含める必要があることが課題であった。そこで、提案手法では、フレームワーク側の振る舞いを、そのコマンドの粒度で抽象化しライブラリ化することで、フレームワーク部分の扱いを簡略化する。この分割により静的検証の対象となるソースコードの量を大幅に低減できるため、静的検証作業を高速化できる。

モデル駆動型開発手法のセキュリティ保証への拡張は、セキュリティ要求と実

装をモデルを介してつなぐことで、セキュリティの問題への取り組みを自動化するものである。ただし、モデルの作成とコードの実装が、独立した開発プロセスであるかぎり、モデル作成は要求定義の文書化と同様に、開発者にとっては冗長的な作業であり、アジャイルソフトウェア開発手法との整合は低いものであった。本提案手法では、コードからのモデルの逆生成を行うことで、モデル上でのセキュリティチェックを実現する。制御フローやデータフローに個々のコマンドがどう関連するかをコマンド抽象化ライブラリに登録することで、コードからのモデルの逆生成を実現する。この定義を用いることで、開発部分のアプリケーションコードの静的検証から効率的にアプリケーションの振る舞いモデル（制御フローモデルとデータフローモデル）の生成が可能となる。ここでいうモデルは、アプリケーションの振る舞いを示す状態遷移グラフ（制御フロー）と外部とのデータの授受を追うためのデータフローグラフの２つである。Web アプリケーションにおける制御フローは、ブラウザ側とサーバー側の Web アプリケーションとのインタラクションに伴うページ遷移を示す。生成した制御フローモデルはセキュリティ機能の検証に用いる。データフローモデルはインジェクション攻撃の検証に用いる。

アジャイルソフトウェア開発手法では、開発の中心はテストケースの作成と、そのテストをパスするコード実装作業である。したがって、本提案手法のように、セキュリティ保証に必要十分なレベルで、最も詳細な情報を持つ実装コードからモデルを抽出する方法が現実的である。

セキュリティに関係するコマンドは (セキュリティ要求に基づいた) セキュリティ機能を実現するコマンド（Security Command, SC）と、その使用がセキュリティ上のリスクとなる可能性のあるコマンド (Risky Command, RC) の二種類に分類する。アクセス制御のようなセキュリティ機能は、アプリケーションの設計の一部であり、これには主に SC で対応する。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のような Web アプリケーションで一般的な問題は、フレームワーク側で対応済みであるが、脆弱性として問題となる一例は、フレームワークが提供するセキュリティ機能を無効化して、アプリケーション側での独自機能を実装する場合である。こうした問題には、セキュリティ機能の無効化コマンドを RC として定義することで対応する。これにより、設計と実装双方の問題にコマンド抽象化ライブラリで対応可能となる。セキュリティテストのテストカバレッジの計測にもコマンド抽象化ライブラリが活用できる。プログラム中における上記の SC,RC の配置を「SINK」として定義し、この SINK に対するテストカバレッジをセキュリティ・テストのテストカバレッジとして集計する。個々のセキュリティ機能 (SC) については、その配置の網羅性はモデル上で静的に行い、サンプリングによる最小限のテストケースで振る舞いを確認する。RC の使用や、静的検証で指摘された問題（False-Positive、擬陽性）については、それぞれにテストケースを作成し、問題が無いことを確認する。テストカバレッジが計測できることで、戦略的なセキュリティテストの配置が可能となる。

開発者は (暗黙もしくは明示双方の) セキュリティ要求に基いてアプリケーションを開発し、ソフトウェア開発に関する、共通で最新のセキュリティ知識にもとづいてセキュリティ保証を実施する必要がある。本研究では Common Weakness Enumeration (CWE) をコマンド抽象化ライブラリを用いて、各コマンドに対応付けることで、開発者がアプリケーションコード上に現れるセキュリティ関連コマンドの特性を、体系的に捉えることができるようになる。つまり、実装コードからセキュリティ要求とその対策やテストの関係などの知識にリンクすることで、体系的で網羅的なセキュアなアプリケーション開発を補佐する。このように、実装コードから、関連する脆弱性情報、推測されたセキュリティ要求を抽出する。最終的な、セキュリティ要求の確認は開発チームのレビューが必要であるが、セキュリティ要求の保守作業をプロセス化し、作業負担を低減することが可能になる。こうした開発のプロセスフローをツールがサポートすることで、反復開発に伴う要求や実装の変更による脆弱性の発生を未然に検知できるようになる。例えば、前回の定義と比較することで、実装の変更が要求定義の更新を伴うものなのか、またはセキュリティ要求の変更が実装コードの変更を伴うべきものなのかを自動的に確認できる。

このように、コマンド抽象化ライブラリを中心に、セキュリティに関する情報を集約することで、実装コードからアプリケーションの振る舞いモデル生成、セキュリティ要求の抽出、静的セキュリティ解析、動的セキュリティテストカバレッジの計測の自動化が可能となる。提案手法の評価にあたり、次の 3 つの目標を設定した。1) 要求や設計に起因するセキュリティの問題と、実装に起因するセキュリティの問題とを統一した手法 (ツール) で取り扱えること。2) コマンド抽象化ライブラリの作成保守が大きな負担にならないこと。3) アプリケーション付随の回帰テストでセキュリティ保証を行える (テスト駆動開発にセキュリティテストが組み込める) ことである。これらの目標が実現できることを、アジャイル型開発を代表する Web アプリケーションフレームワーク、Ruby on Rails を対象としたセキュリティツールの開発を通して提案手法の有効性について確認する。開発したツール、RailroadMap は、オープンソースとして公開[11] することで、広く本研究の成果を実際の開発に普及させるよう取り組んでいる。

本研究の貢献は次の 4 つである。

1. コマンド抽象化ライブラリを用いた Web アプリケーションフレームワークのセキュリティ保証手法の提案。

2. Web アプリケーションのセキュリティ問題を扱うモデル定義 (セキュリティ検証モデル) の提案。

3. SINK を用いたセキュリティテストカバレッジの計測。

[11] https://github.com/munetoh/railroadmap

4. 実装コードとセキュリティ知識との自動連携。

5. 提案手法のツール化（RailroadMap の開発）。

1.4 本論文の構成

2 章では、本研究の技術的背景と関連研究について述べる。まず、開発プロセスとセキュリティ保証プロセスの関係を整理し、アジャイルソフトウェア開発に適したセキュリティ保証の取り組みについてその課題を整理し、Web アプリケーション開発に関するセキュリティ技術および従来のセキュリティ保証手法について述べる。次に、モデル駆動開発とセキュリティ保証の取り組みと、今回実装対象とする Rails について述べ、最後に課題について整理する。3 章では、提案手法であるコマンド抽象化ライブラリを活用したセキュリティ保証の詳細について述べる。4 章では提案手法のツール化、Rails に対応したセキュリティテストツール、RailroadMap の機能及び実装の詳細と、RailroadMap を実際の Web アプリケーションに適用し、提案手法の有効性及び実効性について実験した。5 章では本提案手法の有効性についてまとめるとともに、今後のセキュリティ保証のあり方について議論する。6 章で本論文をまとめる。

第2章　技術背景及び関連研究

本章では、本研究で対象とするアジャイルソフトウェア開発でのセキュリティ保証と、Web アプリケーションにおけるセキュリティ保証の取り組みに関してその背景、及び関連研究についてまとめるとともに、従来技術の課題について整理する。

最初に 2.1 節でソフトウェアのセキュリティに関する知識の共有化についてまとめる。次に、2.2 節で Web アプリケーションに関するセキュリティ上の問題についてまとめる。一般にセキュリティの問題は様々な原因から発生するため、本論文で扱うセキュリティの問題について、その範囲と種類について定義する。セキュリティ保証の様々な手法は、実際には開発プロセスに深く依存しており、開発プロセスが変化した場合に、従来の手法の適用が困難になることが指摘されている。そこで、2.3 節では、従来のソフトウェア開発におけるセキュリティ保証の取り組みを整理した後、アジャイルソフトウェア開発におけるセキュリティ保証の取り組みと課題について整理する。本研究の提案手法は、モデル駆動開発によるセキュリティの取り組みを拡張している。そこで、2.4 節で、従来のモデル駆動型のセキュリティ保証の取り組みをまとめ、その課題について整理する。次に、2.5 節で本研究で実装のターゲットとする Ruby on Rails についてその概要を説明する。最後に 2.6 節で本研究で取り扱う課題についてまとめる。

2.1 セキュリティ知識

この節では、一般的なセキュリティ要求と、各種のセキュリティ知識データベースについてその概要を説明する。

ウォーターホール型の開発でのセキュリティ保証は、1) セキュリティ要求が正しく把握され、2) その要求に基づいたセキュリティ対策が設計と実装において反映され、3) 最終的にテストを通して確認される。アジャイルソフトウェア開発においても基本は同じだが、要求、設計、実装それぞれが反復開発のなかで変化してゆくことが大きな違いである。特にアジャイルソフトウェア開発では 1 の文書化がおろそかになりがちである。この場合セキュリティ要求定義が曖昧なため 2 や 3 のセキュリティ保証やテストの実施が不十分となる。ただし、この場合もセキュリティの要求は暗黙の了解として存在するはずであるが、開発者の中で正しく共有されているとは限らない。こうしたセキュリティ要求の定義や保守を効率的に行うためには、共通のセキュリティ要求の定義や脆弱性の定義が必要になる。

2.1.1 セキュリティ要求

一般的なセキュリティ要求を定義するガイドラインには、米国政府が定めた IT システムに関する最低限のセキュリティ要求、FIPS PUB 200[5] や、クレジットカード業界が定めた Payment Card Industry Data Security Standards[4] 、OWASP[1] (Web アプリケーションのセキュリティを扱う NPO) が定めた Web Application Security Requirements [2] などがある。FIPS 200 ではセキュリティ評価要求項目の一つに挙げられているが、情報システムのセキュリティ評価を行う国際標準としては、ISO/IEC15408 (コモンクライテリア) がある [40]。特に、ISO15408 におけるセキュリティ機能定義ではセキュリティ機能とその依存性が体系的に整理されている。

セキュリティ要求には運用など様々な観点が含まれるが、表 2.1 で上記 4 つの要求項目 (15408 については機能項目) を一覧にする。アプリケーションドメインに特化したセキュリティ要求を参照することで、より詳細なレベルのセキュリティ要求が得られる。一般に、個別のアプリケーションに必要とされるセキュリティ要求やセキュリティ機能は脅威分析により導き出されるが、以上のように、政府や業界で策定されたセキュリティ要求（ISO15408 の場合は Protection Profile）を参照することで、統一的なセキュリティ要求を作成することができる。Web アプリケーション開発において、開発者が参照すべきセキュリティ要求は、OWASP のセキュリティ要求定義が基本となるが、Web アプリケーション自体のセキュリティ要求は、例えば E コマースサイトであれば、PCI–DSS に対応する必要がある。

[1] https://www.owasp.org/

表 2.1: セキュリティ要求

	FIPS PUB 200	PCI DSS	OWASP Security Req.	ISO/IEC 15408 part 2
Authentication	✓		1	FIA
Password rule		2	1.3	FIA_SOS
Access Control	✓	7,8,9	2	FDP
Session Management			3	FTA
Parameter			4	
Output			5	
Communication protection	✓	4	6	FTP
Cookie			7	
Web browser			8	
Audit	✓	10	9.7	FAU
Cryptography			9.2	FCS
Privacy				FPR
Contingency	✓			FRU
Configuration management	✓		9.1, 9.5	
Test		11		
Firewall		1		
Data protection		3		
Anti virus		5		
System integrity	✓			
SDL		6		
Certification	✓			
Policy		12		
Incident responce	✓			
Maintenance	✓			
Physical protection	✓			
Planning	✓			
Personnel Security	✓			
System Acquisition	✓			
Training	✓			

2.1.2 セキュリティ知識 (脆弱性と攻撃パターン)

ソフトウェアの脆弱性を把握し共通の知識とする取り組みとして、各種のセキュリティ知識データベースの整備がある。ソフトウェアの脆弱性が社会問題となり、その対応として脆弱性のトラッキングが 1999 年に始まった。これが Common Vulnerabilities and Exposures (CVE、共通脆弱性識別子) である。問題があるソフトウェアのバージョン、問題の種類、対応策をまとめたものである。その後 2005 年に問題の深刻度を定量化する仕組みとして、Common Vulnerability Scoring System (CVSS、共通脆弱性評価システム) が開発された。また、脆弱性の分類として 2006 年に Common Weakness Enumeration (CWE、共通脆弱性タイプ一覧) が、攻撃パターンの分類として Common Attack Pattern Enumeration and Classification (CAPEC) が 2008 年に開発された。これらは絶えず更新されかつ公開されており、ソフトウェアのセキュリティ問題を扱うための共通の知識とし

て参照可能である。セキュリティ保証の自動化を行うためには、このようなセキュリティ知識の分類とデータベース化が重要である。

これらは米国政府が Federal Information Security Management Act (FISMA)[2] の元に Security Content Automation Protocol （SCAP) [3] の取り組みの一環として整備を進めているものであり、最終的にはソフトウェアの脆弱性管理の自動化を目指すものである。これらのデータベースは非営利団体の米国 MITRE 社 [4] により管理されている。

次に、本提案と関係の深い CVE、CWE、CAPEC についてその概要を説明する。

2.1.2.1 Common Vulnerabilities and Exposures (CVE)

CVE はソフトウェアで発生した脆弱性の記録である。MITRE 社により重複がないように管理されている。日本国内では Japan Vulnerability Notes(JVN)[5] が CVE と連携して脆弱性情報を管理している。

CVE で登録された脆弱性がセキュリティの観点でどのくらい深刻なものなのかを定量化する手法として Common Vulnerability Scoring System（CVSS）[6] が策定されている。脆弱性の影響度を数値化することで、対策の緊急度を知ることができるようになる。

脆弱性情報を一元管理することで、ソフトウェアの利用者は該当する脆弱性情報を参照しやすくなる。Web アプリケーションの開発者の場合も、利用しているフレームワークや外部パッケージに脆弱性が見つかった場合に脆弱性のアプリケーションへの影響を確認する。問題があれば直ちに利用コンポーネントの更新作業を行う。

2.1.2.2 Common Weakness Enumeration (CWE)

CWE はソフトウェアにおける脆弱性の種類を識別するための共通の基準である。2008 年に最初のバージョンが公開され、その後定期的に更新されている [7]。CWE には 4 つのタイプがある。View はある視点で脆弱性を集めたもの、Category は共通の特性をもつ脆弱性のグループ化、Weakness は個別の脆弱性で、さらに Class, Base, Variant の属性でその抽象度が示される。Compound Element は複数の要因が重なって発生する脆弱性を示す。CWE ではこうした情報に一意の番号が割り当てられ、それが相互にリンクして階層的な構造になっている。CVE で登録された脆弱性もその種別を CWE で指定される。

[2] http://csrc.nist.gov/groups/SMA/fisma/
[3] http://scap.nist.gov/
[4] http://www.mitre.org/
[5] http://jvn.jp/
[6] https://www.first.org/cvss
[7] 本研究では Version2.8 を利用した

セキュリティテストツールが発見した脆弱性を CWE を用いて特定することで、関係者が共通の知識の元に正確に脆弱性情報を参照し、対策を行うことができるようになる。

2.1.2.3 Common Attack Pattern Enumeration and Classification (CAPEC)

CAPEC は攻撃パターンを整理したデータベースで、上述の CWE とも相互リンクされる。CAPEC も CWE 同様に、3 つのタイプとともに階層構造で体系化されている。View はある視点で攻撃を集めたもの、Category は共通の特性をもつ攻撃のグループ化、Attack Pattern は個別の攻撃パターンとなる。

CAPEC の情報は CWE とリンクされる。アプリケーション開発者から見ると、こうした情報はテストケースの作成に有効である。

2.1.3 セキュリティ知識に関する関連研究

Wang らは CVE や CWE などの知識をセキュリティのオントロジーとして利用することで情報システムの潜在的な脆弱性を類推している [65] [66] [67] 。これは過去に発生したセキュリティ問題を元にそのシステムのセキュリティ上のメトリックスを求めている。ここで用いるメトリックスは CVSS 元に計算される。

Almorsy らはアーキテクチャレベルでのセキュリティ分析に CAPEC の攻撃シナリオを利用している [10] [11]。対象となるソフトウェアを独自の XML を使ってモデル化し、OCL[8] を用いて定義した CAPEC の攻撃シナリオを用いて検査することで、攻撃の可否を判定する。

どちらの研究も上位のセキュリティ分析にこうした知識を活用しているが、開発者が実装をすすめる中で必要なセキュリティ知識を参照できるようにする仕組みには至っていない。

[8] Object Constraint Language, UML モデルに適用する規則を記述するための宣言型言語

2.2 Webアプリケーション開発とセキュリティ保証

この節ではWebアプリケーションの概要とセキュリティの問題を整理する。

2.2.1 Webアプリケーションと脆弱性

1991年に欧州原子核研究機構(CERN)のTim Berners-LeeがWebサイトを公開した後、Web技術はインターネット上で急速に普及した。初期のWebは、プロトコルにHTTP、コンテンツの記述にマークアップ言語のHTMLを用いて、静的なコンテンツをサーバーからクライアントに送るだけのものであったが、CGIの登場により、クライアントからのリクエストでリソースが動的に生成できるようになる。これがWebアプリケーションである。その後、W3Cによる標準化の元、様々な技術がウェブに組み合わされ、その機能が拡張されてきた。

Webアプリケーションは、ネットワークを介して使用される分散アプリケーション・ソフトウェアである。現在、Wiki、Blog、ゲーム、オンラインショッピング、オンラインバンキングなど、様々な目的のアプリケーションがインターネット上で運用されている。一般的には、Webサーバーと Webブラウザーで構成され、様々な実装形態があるが、基本となるのは、HTMLプロトコルを利用したクライアント・サーバーシステムである。

Webアプリケーションは、インターネットという公開されたネットワークで実現する、分散アプリケーションであり、さまざまなセキュリティ上の問題が、インターネットおよびWeb技術の開発初期から指摘されている。また新しい技術や機能の導入が盛んな分野であり、それに伴って絶えず新しい脆弱性や脅威が現れているのが実情である。しかしながら、Web技術は社会インフラとして広く利用されており、また主要な攻撃(サイバーアタック)の対象でもある。したがって、その安全性の確保は重要な課題である。

Webアプリケーション固有の脆弱性として、クロスサイトスクリプティング(XSS)、SQLインジェクション などがある。マイクロソフト社が2003年に示した脆弱性の分類[51]を表2.3に示す。

こうした問題は、Webアプリケーションを実装する言語やフレームワークによって、脆弱性の発生頻度が異なることが指摘されている[34][56]。当然、Webアプリケーションのもつ固有の脆弱性に対応した最新の言語やフレームワークを用いることで、開発者のセキュリティ対策の負担は大きく低減する。

2.2.2 Webアプリケーションフレームワーク

Webの仕様や機能が複雑になるにつれて、その開発は難しさを増してゆく。そこで、Web開発で必要となる、基本機能をサポートしたフレームワークが開発さ

れ、そうしたフレームワーク上に Web アプリケーションを構築する手法が一般化してゆく。

図 2.1 はその変遷を示したものである。PHP(Hypertext Preprocessor) は、動的に HTML データを生成するためのプログラミング言語で、現在も広く利用されている。JSP(Java Server Pages) は HTML に埋め込まれた Java コードを使い、動的に HTML データを生成する技術であり、サーバーで稼働する Java Servlet とともに Java による Web アプリケーションを実現するための主要な技術である。EJB(Enterprise Java Beans) は、ネットワーク分散型ビジネスアプリケーションの仕様であり、業務アプリケーションの開発に広く利用されている。Web アプリケーションにおいても、MVC(Model-View-Controller）型のアーキテクチャとして、 Apache Struts フレームワークや 軽量な Spring MVC フレームワークが開発された。

2004 年に David Heinemeier Hamsson により、 Ruby on Rails の最初のバージョンが公開された。Ruby on Rails は、スクリプト言語である Ruby を用いた MVC アーキテクチャの Web アプリケーションフレームワークであり、これまでのフレームワークに比べ、大幅に少ないコードで簡単に Web アプリケーションの開発ができることが大きな特徴である。Django は 2005 年に公式リリースされた Python で実装した Web アプリケーションフレームワークで、Rails 同様に Web アプリケーション開発を軽量にすることができる。CachePHP は、Rails の影響を受けて開発された PHP ベースの Web アプリケーションフレームワークで、2005 年に最初のバージョンがリリースされた。

2000 年台前半の Web アプリケーション開発では、Java が多く用いられており、開発スタイルも Waterfall 型が主流であった。2004 年以降、Rails などの普及とともに、Web アプリケーション開発でも、アジャイルな開発手法が取り入れられるようになってきた。これは、保守すべきアプリケーションのコード量が激減し、またフレームワークの機能が様々な自動化によって充実した結果、少人数で大規模なアプリケーション開発が可能になったためだと考えられる。表 2.2 にこうしたフレームワークで構築されたサイトの一例を示す。大規模なサイトで広く実際に利用されていることがわかる。また、こうした実サイトでの利用がフレームワークの進化を加速している。

2.2.3 Web アプリケーションのセキュリティ保証手法

Web アプリケーションの開発で主に用いるセキュリティ保証手法には、ソフトウェアの実装コードを静的に解析して、セキュリティ上の問題を解析する、静的検証（静的テスト）と、実際にアプリケーションを動作させて確認を行う動的テストがある。また、その双方を組み合わせた形のテスト手法も開発されている。これらのテストについてはツールの形で実用化が進んでおり、実際の開発や運用で

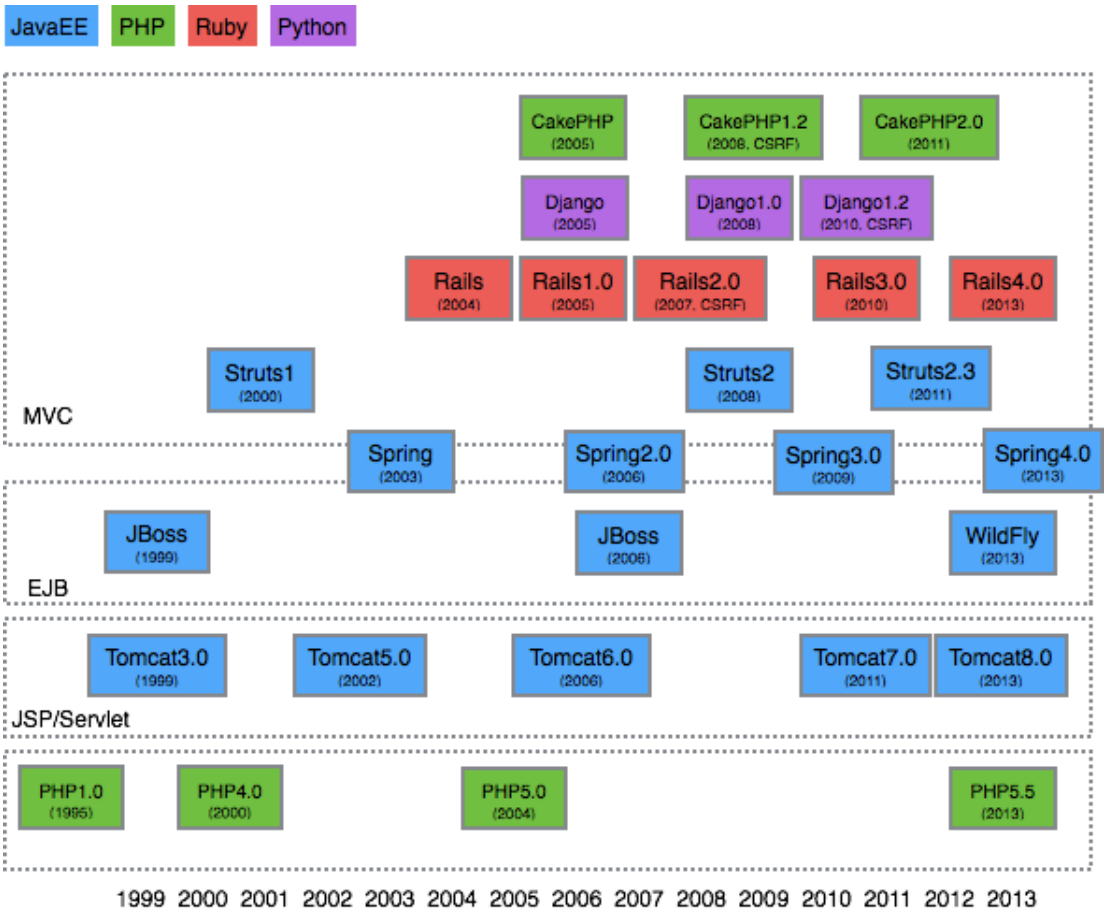

図 2.1: Web アプリケーションフレームワークの歴史

広く利用されている。

より設計寄りの観点からセキュリティ保証を実現する方法としては、モデル駆動設計をセキュリティにも拡張する手法の研究が進められている。これについては 2.4.2 で扱う。

2.2.3.1 脅威分析

脅威分析には様々な手法がある。一般的には、1）資産の特定、2）資産への脅威の洗い出し、3）リスク分析 の 3 段階である。その結果、セキュリティ対策を検討し、セキュリティ機能を設計し実装する。また、実装に起因する脅威として、Web アプリケーション固有の脆弱性への対応も含まれる。アジャイルソフトウェア開発においても、ミスユースケース [9] を用いた脅威分析によるセキュリティ要求の獲得が提案されている [21][33]。こうした脅威分析は前提となる要求が変化する都度実施する必要があり、アジャイルソフトウェア開発では軽量な脅威分析手

[9] アビュースケース (Abuse case) とも呼ばれる

表 2.2: 主要な Web サイトと利用フレームワーク

Web サイト	タイプ	採用フレームワーク
Twitter (2006-)	ミニ Blog	Rails
Githib (2008-)	レポジトリ	Rails
SlideShare (20XX-)	スライドホスティングサービス	Rails
Cookpad (2008-)	レシピコミュニティーサイト	Rails
Tabelog (2007-)	グルメサイト	Rails
Instagram (2010-)	SNS(写真共有)	Django
Pintrest(2010-)	SNS(写真共有)	Django
OpenStak(2010-)	管理コンソール	Django

法が必要である。

2.2.3.2 静的検証（静的テスト）

Web アプリケーションにおいても、ソースコードの静的解析により脆弱性の発見が可能である。ツールの存在の有無は、Web アプリケーションの実装言語やアプリケーションフレームワークにも依存するが、そうしたツールの存在の有無は、Web アプリケーションの実装方法を選択する際の指標となるべきである。

一般的な、Web アプリケーション向けの静的解析ツールは高速に実行できるため、アプリケーション開発者の負担は少ない。そのため、アジャイルソフトウェア開発との整合性も高い。ただし、静的テストの報告には擬陽性 (False–Positive) が多く含まれる場合がある。これは、静的テストでは疑わしい問題がすべて報告されるためであり、一般に、問題の検出精度と速度の関係にはトレードオフがある。警告が大量に出る場合、それが本当の脆弱性か否かを確認する作業負担が問題となる。こうした作業を低減するためには、適切なコーディング規約を設けて、プログラマが事前に実装スタイルに配慮することで、ツールが報告する警告数を少なくする工夫などが必要である。

セキュリティの問題が、コードの局所的な記述に起因する場合は、静的テストは非常に有効であるが、複数のクラスやメソッドなどの幅広いコードにまたがる振る舞いやデータフローに起因する脆弱性の検知は難しい。また、アクセス制御などのセキュリティ機能の実装に近いセキュリティの問題には対応できない。

また、利用しているフレームワークなどに存在する、既知の脆弱性（CVE で公開済み）については、そのバージョンの確認でチェックすることが可能であり、各種の自動確認ツールが利用可能である。

2.2.3.3 動的テスト（ペネトレーションテスト）

Web アプリケーションに脆弱性が含まれないことを確認する最終的な手段は、ペネトレーションテストである。

ペネトレーションテストでは、実際に稼働する Web アプリケーションに対して様々な攻撃を加えることで 、Web アプリケーションが既知の攻撃に対して充分な耐性があることを確認する。ペネトレーションテストの実施方法としては、手動で稼働する Web アプリケーションにアクセスして、各種の攻撃を試みる場合と、脆弱性スキャナーと呼ばれる、専用のツールを用いて、自動的に実施する場合とがある。カード業界のセキュリティ標準 PCI-DSS でも、専門業者による脆弱性スキャナーによる定期的な Web アプリケーションの検査を要件としている。脆弱性スキャナーは Web サイトをクロールし、すべての入力に対して、様々な攻撃を機械的に試みるため、一般にその実行時間は長い。また、Web サイト上で問題箇所を指摘するため、コードでの修正には手間がかかる。そのため、開発者が頻繁に利用する性質のツールではない。

2.2.3.4 開発者の役割

Web アプリケーション全体のセキュリティ保証は、アプリケーションのソースコード、アプリケーションの設定、Web アプリケーションフレームワークの脆弱性管理、OS の構成と脆弱性管理、ファイヤーウォールや（SSL,HTTPS などの）負荷分散装置など多岐にわたる。

Meier らがまとめた Web アプリケーションの脆弱性（セキュリティ機能でもある）を表 2.3 に示す [51]。この分類に Rails のアプリケーション開発者の責任を次の 3 つの分類で対応付ける。◎ 開発者が責任をもって対応しなければならないセキュリティ機能、○ フレームワークで対応済みであるが、開発者が独自の機能を用いる可能性があるセキュリティ機能、△ その他、フレームワークや外部パッケージで対応済みのセキュリティ機能。なお、アプリケーションが利用しているフレームワークやパッケージの脆弱性管理（もしくは構成管理）は運用面で非常に重要である。

Web アプリケーションに関する脆弱性についても、フレームワークの進歩とともに、開発者への負担は低減している。表 2.4 にセキュリティに関する対応を開発者が行うべきか (Coding)、フレームワーク側で対応済みか (Default) をまとめる。最新のフレームワークでは、多くの脆弱性への対応がフレームワーク側で実装されており、アプリケーション開発者の負担が低減している。

表 2.3: アプリケーションの脆弱性分類 [51]

脆弱性分類	対策	Rails
入力	ユーザー入力データの検証。 XSS, CSRF、SQL インジェクション対策	○
認証	ユーザー認証の採用	◎
認可	ACL、RBAC の採用	◎
構成管理	管理者権限	○
データ	データ保護（認証情報、個人情報）	○
セッション管理	適切な Cookie の使用と管理、リプレイ攻撃対策	△
暗号	適切な暗号の利用、鍵管理	△
パラメーター操作	Cookie など 各種の入力パラメータの検証	△
例外処理	適切な例外処理	△
監査、ログ	適切な監査とロギング。機密情報の排除	△

表 2.4: Web アプリケーションの脆弱性と開発者の対応方法

Weakness	CWE	CGI	EJB	Rails2	Rails 4
XSS	79	Coding	Coding	Coding	default
CSRF	352	Coding	Coding	Coding	Config
Authentication		Coding	Coding	Coding	Coding
Authorization (RBAC)		Coding	Coding	Coding	Coding

2.2.4 Web アプリケーションのセキュリティテストに関する関連研究

Web アプリケーションを静的にテストする手法は広く研究、提案されている。表 2.5 に主な研究についてその手法を整理する。

Huan らは静的検証により Web アプリケーションの脆弱性を検査する手法を提案している [38]。Letarte らは PHP コード (データベースのステートメント) の静的検証で、シンプルなロールモデルを生成した [48]。Dalton らは、 ユーザー固有のアクセス制御モデルを用いて、動的な振る舞いを検証する Nemesis という名前のツールを提案した [25]。Halle らは、Web アプリケーションを状態遷移でモデル化し、ランタイムの動作をテンポラルなプロパティでチェックする手法を提案した [37]。これらの手法は、開発者の膨大な作業を前提としている。Son らは、DB へのアクセスを元にした静的検証によりアクセス制御のチェックを行うツール、Rolecast を提案した [57]。Sun らは、フォースブラウジングによるアクセス制御の脆弱性検出を行う手法を提案した [59]。Alalfi らは 動的な PHP Web アプリケーションから、アクセス制御を SecureUML モデルとして生成する手法を提案して

いるが、検証は人手となる [9] [8]。Gauthier らは PHP Web アプリケーションの、アクセス制御の静的検証を、[59] に比べてより多くの問題を検出し、かつ高速なツール、 ACMA (Access Control Models Analyzer) を開発した [36]。

表 2.5: Web アプリケーションの静的テスト手法の研究

著者 (年)	対象言語、Framework	検出できる脆弱性	特徴
Huan (2004)[38]	PHP	XSS, SQL inj.	WebSSARI の提案
Jovanovic(2006)[41]	PHP	XSS	Pixy の提案
Wassermann(2008)[68]	PHP	XSS	インフォメーションフロー解析
Letarte(2009) [48]	PHP	アクセス制御	シンプルなロールモデルを生成による検証
Dalton(2009)[25]	PHP	アクセス制御	ユーザー固有のアクセス制御モデル
Halle(2010)[37]	PHP	Nav. error	状態遷移モデルを用いたランタイム保護
Son (2011) [57]	PHP	アクセス制御	DB へのアクセスから検証
Sun (2011) [59]	PHP	アクセス制御	フォースブラウジングの検証
Gauthier(2012)[36]	PHP	アクセス制御	Sun の手法 [59] を高速化したツール ACMA (Access Control Models Analyzer) を開発
Chaudhuri(2010)[23]	Ruby on Rails	XSS	シンボリックエグゼキューション
Doupe(2011)[28]	Ruby on Rails	EAR	シンボリックエグゼキューション
Collins(2012)[24]	Ruby on Rails	XSS	高速で実用的な Brakeman

Web アプリケーションを動的にテストする手法はツールとしての製品化が進んでおり、その性能評価 (精度、実行速度、使いやすさ) に関して議論が行われている [60] [64] [61] [16] [30] 。脆弱性スキャナの評価に関する研究を表 2.6 にまとめる。実施者の熟練度やツールの性能差により、大きく結果が異なる問題が指摘されている。また、脆弱性スキャナーを用いた場合のテストカバレッジの定量化が難しい。脆弱性スキャナーの性能向上に関しては、状態遷移モデルを構築し、Fuzzing を用いて効率化する手法 [29] が提案されている。

Web アプリケーションのセキュリティテストのカバレッジ計測の研究は少ないが、Dao らはセキュリティに関する関数を SINK と定義し、その解析からテストカバレッジを求める手法を提案している [26][27]。

表 2.6: Web アプリケーションの動的テスト手法の研究

著者 (年)	内容
Suto(2007)[60]	脆弱性スキャナの性能差を指摘
Vieira(2009)[64]	300 の WebApp の脆弱性を調査
Suto(2010)[61]	7 種類の脆弱性スキャナを評価
Bau(2010)[16]	8 種類の脆弱性スキャナを評価
Doupe(2010)[30]	11 種類の脆弱性スキャナを評価
Doupe(2012)[29]	Fuzzing を用いた効率化

2.3 アジャイルソフトウェア開発とセキュリティ保証

この節では、ソフトウェア開発プロセスと、セキュリティ保証について、従来技術と課題についてまとめる。最初に、2.3.1 節で従来のウォーターフォール型のソフトウェア開発とセキュリティ保証の取り組みを整理する。次に、2.3.2 節でアジャイルソフトウェア開発の概要を説明した後、アジャイルソフトウェア開発におけるセキュリティ保証の取り組みと未解決の課題について整理する。

2.3.1 ウォーターフォール型のソフトウェア開発（V 字型モデル）とセキュリティ保証の関係

ソフトウェア開発工程の標準は ISO/IEC 12207[39] で定義されている。これはウォーターフォール型（または V 字型モデル）をベースとしており、その概要を図 2.2 に示す。この図では、角丸長方形 (グレー、Rounded Rectangle) がプロセス、長方形 (白色、Rectangle) が文書やコードなどの成果物を示す。計画型のソフトウェア開発では、企画 [1]、要求定義 [2] 、設計 [3] 、実装 [4] 、ユニットテストと結合テスト [5]、受け入れテスト [6] の各段階を経て最終製品がリリースされ、稼働する [7]。この手法は大規模なソフトウェア開発やシステム開発で広く用いられている。計画型のソフトウェア開発では、実装するソフトウェアが膨大であるため開発コスト（人員、期間）を要する。要求定義や設計のミスによる修正の発生はプロジェクトの成否を左右するため、できるだけ正確な要件定義と設計を行い、開発リスクの低減をはかる必要がある [20] 。

情報システム開発におけるセキュリティ対策は、“NIST SP800-64 Security Considerations in the System Development Life Cycle (情報システム開発ライフサイクルにおけるセキュリティの考慮事項)” で, ガイドラインとしてまとめられている [54]。NIST SP800-64 では 、SDLC(Systems/Software Development Life Cycle) に沿う形で、セキュリティ確保のために実施すべき取り組みを整理している。

図 2.2 に対して、セキュリティ保証の取り組みと成果物をマップしたのが図 2.3 である。この図では、角丸長方形 (黒、Rounded Rectangle) がセキュリティ・プロセス、長方形 (黒色、Rectangle) が文書などの成果物を示す。まず上流工程では、脅威分析 [8] を行う（アブユースケース、ミスユースケースなどを用いて資産を特定し、脅威を洗い出す)。そうしてセキュリティ要求を定義する。次に、セキュリティ要求を満たすためのセキュリティ機能を洗い出し、設計に反映させる。こうした作業には専門家の参加が必要であり、ISO/IEC15408 (CC: Common Criteria)[40] で必要とされる文書、Security Target (ST) や Protection Profile (PP) を作成するのに相当する労力を要する。CC はセキュリティ製品における政府調達の統一基準として運用されている国際標準で、評価対象の製品は脅威分析と必要なセキュリティ対策を文書化した ST や PP を作成し、第三者の評価機関からセキュリティ

認証を得る枠組みである。

次に、実装（コーディング）を行うが、使用するプログラミング言語や外部コンポーネントにより脆弱性の問題は異なる。コーディングの問題については、適切なコーディング規約の採用と、静的検証ツールの利用によるソースコードの検査 [9] が必要である。

最終的には、想定した脅威に対応出来ていることを、実際の攻撃を模したペネトレーションテストや倫理 (Ethical) ハッキングテストを通して確認 [10] した後、最終製品としてリリースする。Web アプリケーションなどでは、脆弱性スキャナーなどの自動テストツールが最終確認として利用することができる。こうした自動ツールによる定期的な検査は 2.1.1 節で紹介した PCI-DSS でも要求されている。

脆弱性を完全に排除することは難しく、新たな脆弱性が見つかった場合、迅速な修正と更新 [11] が重要である。

以上のセキュリティ対応には人員、工数、各種ツールの使用コストが必要であり、開発に付加される。大きなソフトウェア開発会社では専属のチームが対応しているが、社内にそうしたチームを持たない場合には、外部に依頼することになる。BSIMM などの成熟度モデルはこうしたセキュリティに関する取り組みを、組織レベルで評価する。

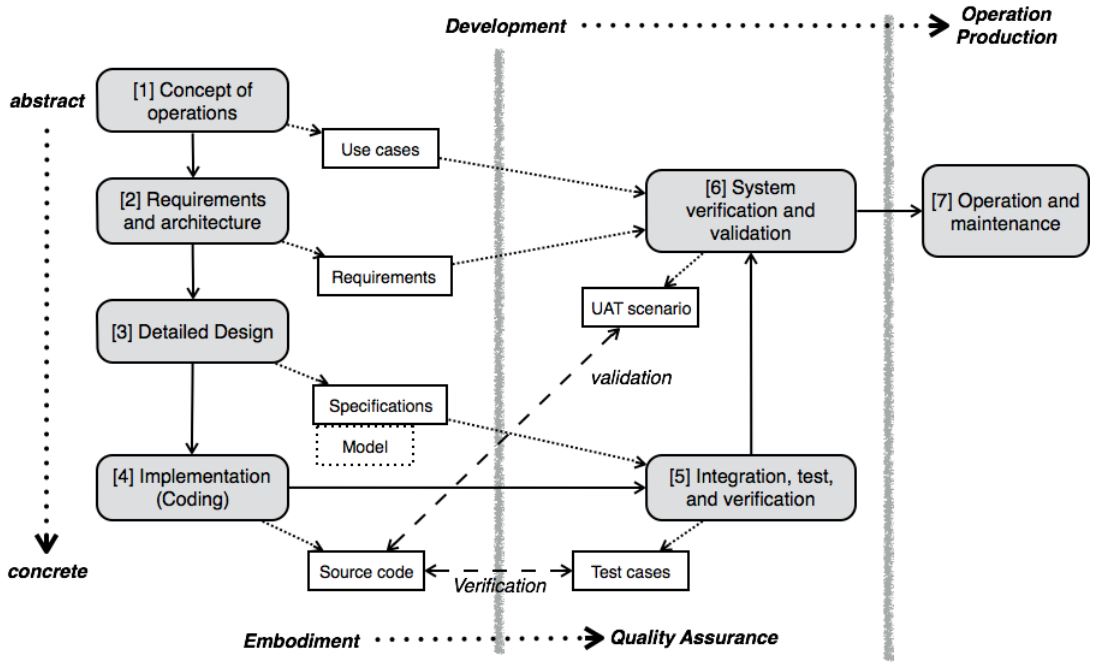

図 2.2: Waterfall 型の開発モデル

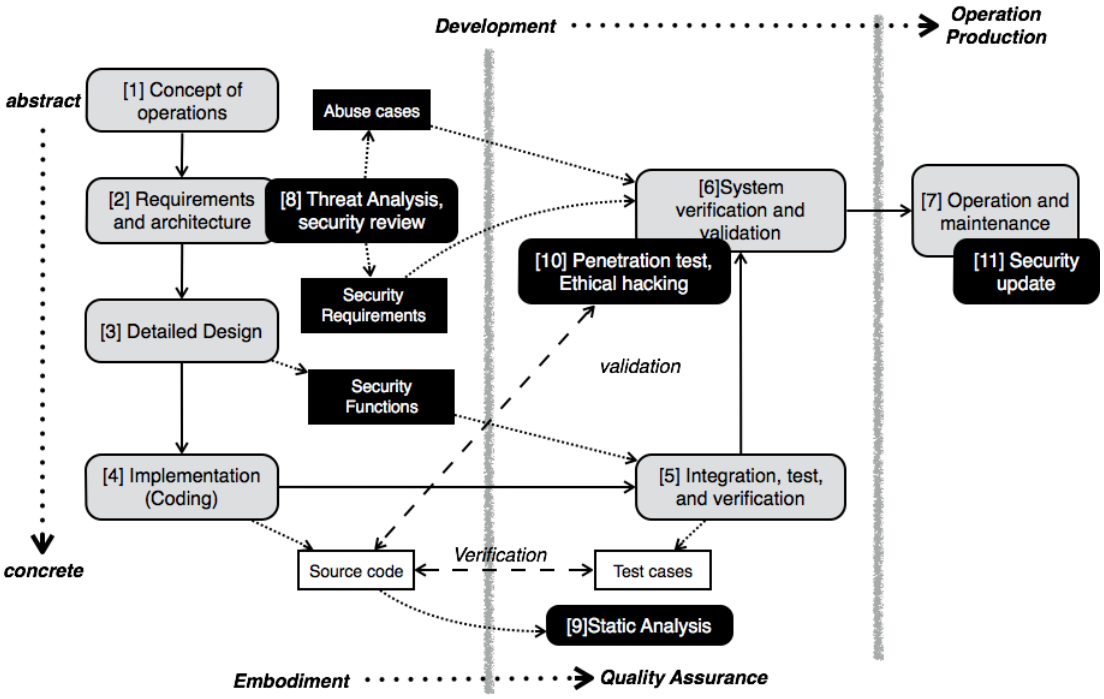

図 2.3: Waterfall 型の開発モデル + セキュリティ保証

2.3.2 アジャイルソフトウェア開発とセキュリティ保証

アジャイルソフトウェア開発とは、迅速かつ適応的にソフトウェア開発を行う手法の総称であり、2001 年に定義されたアジャイルソフトウェア開発宣言[10] を以下に引用する。

アジャイルソフトウェア開発宣言

「私たちは、ソフトウェア開発の実践あるいは実践を手助けをする活動を通じて、よりよい開発方法を見つけだそうとしている。この活動を通して、私たちは以下の価値に至った。
プロセスやツールよりも個人との対話を、包括的なドキュメントよりも動くソフトウェアを、契約交渉よりも顧客との協調を、計画に従うことよりも変化への対応を、価値とする。すなわち、左記のことがらに価値があることを認めながらも、私たちは右記のことがらにより価値をおく。」

アジャイルソフトウェア開発は適応的な開発であり、リリースサイクルも週単位以下と非常に短期間である。これに対して、従来のウォーターフォール型の開発は計画重視の開発であり、数ヶ月から数年の開発期間を想定する。変更コストが増大しないという前提で、事前計画よりも柔軟性を重視する。アジャイルソフトウェア開発においても、計画、要求分析、設計、実装 (コーディング)、テスト、文書化 が実施される。現実的には実装 (コーディング) とテストが主体となり、チーム作業による意思の疎通が重んじられ、結果として文書化が少ない。

アジャイルソフトウェア開発の代表的な手法には下記が知られており、また広く現場でも実践されている。

反復 **(**イテレーション**)** 小さな作業対象、短い開発期間で開発を行う。リスクを最小化しようとしている。

テスト駆動開発 **(TDD: Test-driven Development)** テストファーストデザイン、テストケースを最初に用意し、そのテストが通るようにコーディングを進める。

リファクタリング 外部インターフェースや振る舞いの変更は行わないコードの最適化を行う。

継続的インテグレーション 絶えず稼働する状態をキープする

機能駆動型開発（**FDD: Feature Driven Development**）

また、次のような定式化されたアジャイルソフトウェア開発手法がある。

エクストリーム・プログラミング **(XP: eXtreme Programming)** 5 つの価値、19 のプラクティス [1]

[10] http://www.agilemanifesto.org/

スクラム **(Scrum)** バックログによる要求管理、スプリントとよぶ反復 [3]

表 2.7 にウォーターホール型開発とアジャイル型開発の比較を示す。アジャイル型開発で必要なセキュリティ保証の手法は図 2.4 の開発フローに適合する、高速、軽量な仕組みである。その実行に何日もかかるような脆弱性スキャナではなく、コンパイラのようにコマンド実行で即座に実装とデザインの問題点を指摘するツールが必要である。

表 2.7: ウォーターホール型開発とアジャイル型開発の比較

	ウォーターホール型開発	アジャイル型開発
開発コード量	膨大	少ない
開発者	100 － 1000 名	数名ー 20 名
開発の中心	計画	コード（テスト、ソース）
文書化	充実	不十分
リリースサイクル	半年、数年	週
モデル駆動開発	適	不適
モデル駆動テスト	適	不適
セキュリティテスト手法	確立 (成熟度モデル)	未整備

すべてのウォーターフォール型の開発がアジャイルソフトウェア開発に移行できるわけではない。少なくとも、アジャイルソフトウェア開発の実践にはいくつかの条件がある。小さなチーム、顧客との直接的なコミュニケーションなどが条件としてとりあげられるが、それが可能となるのは、開発し保守すべきコード量が重要であると考えられる。実際にアジャイルソフトウェア開発が実践されている Web アプリケーション開発では、フレームワークの整備が進んでおり、アプリケーションの実装は、その振る舞いを実現するための最小限のコードで実現することが可能である。つまり、フレームワークにより、Web アプリケーションで必須な機能の大半がサポートされており、開発者は個別のアプリケーションのロジックに注力すれば良い。

図 2.4 にアジャイルソフトウェア開発プロセスの一例を示す。この図からわかるように、従来のウォーターフォール型のソフトウェア開発とは開発の流れが全く異なる。これが可能となるのは、アプリケーションドメインに特化したフレームワークの存在が大きい。

この場合、ウォーターフォール型で重要であった上流工程は、フレームワークに吸収されていると考えることができる。フレームワークはセキュリティガイドラインを示すが、それはフレームワークを用いたアプリケーション実装のセキュリティに関する注意点であり、Web アプリケーションの持つべきセキュリティ要求のすべてを規定するものではない。

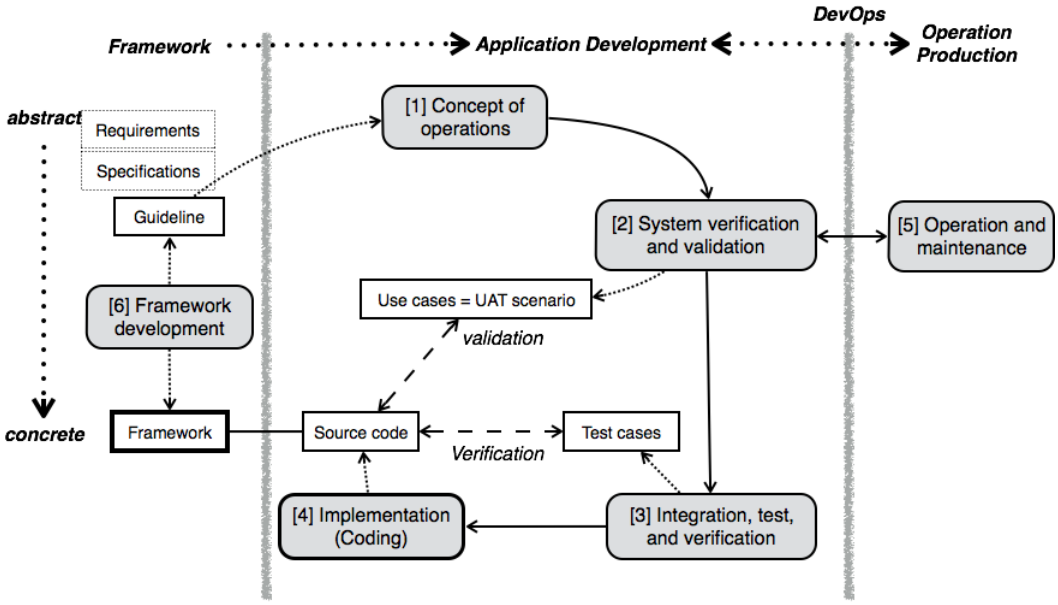

図 2.4: Agile 開発モデル

2.3.3 アジャイルソフトウェア開発のセキュリティに関する関連研究

ここでは、アジャイルソフトウェア開発とそのセキュリティ保証の問題ついて、関連研究をまとめる。

Beznosov らは、従来のセキュリティ保証の手法 26 個のアジャイルソフトウェア開発との整合性を 4 つのタイプ（整合、開発手法に無関係、自動化で対応可能、非整合）に分類した [19]。その中で、明確に不整合と分類された手法は 12 個あり、半数近くの手法がアジャイルソフトウェア開発に適さないとしている（表 2.8 参照）。その後、Keramati はセキュリティ保証手法のアジャイルソフトウェア開発手法への適用性の定量化を試みている。具体的には 9 つの指標（単純さ、カスタマーとのやりとり、モデリングや文書化の負担、変更への耐性、実行速度、非形式性、教育、反復性、柔軟性）について 5 段階で評価することで、ジャイルソフトウェア開発手法との整合性を数値化した [45]。また、Al-Ahmad らは CLASP のセキュリティ保証手法の XP のマッピング方法を提案している [6]。プロセスで見た時のアジャイルソフトウェア開発とセキュリティ保証手法の不整合の原因は、文書化と第三者によるテストやレビューである。

セキュリティの問題に対応できるようにアジャイルソフトウェア開発手法を拡張する提案としては、以下の研究がある。

Nicolaysen らが実施したケーススタディでは、アジャイルソフトウェア開発手法を採用している開発現場において、たとえそれが Web アプリケーションのように攻撃が想定される場合でも、セキュリティの確保にむけた特定の手法が用いられていなかった。そこで、アジャイルソフトウェア開発手法 (ここでは Scrum) に

セキュリティを組み込む方法として２つの拡張を提案している。ひとつはバックログにセキュリティ要求を加えることで、チーム全体でセキュリティの問題を共有すること。もうひとつは、セキュリティ機能の実装をテスト駆動開発で進めることである [53]。

Erdogan らも同様に Scrum に対してセキュリティ保証の手法を追加した手法 EAST(Extended Agile Security Testing) を提案している [33]。EAST ではバックログに対して、ミスユースケースを作成しセキュリティ要求を洗い出し、受け入れテストに反映させる。また、セキュリティに関する判断を文書化し蓄積することでチーム内でセキュリティの知識を共有する。

Bostrom らは XP を拡張し、アブユースをセキュリティ要求を獲得し利用する手法を提案している [21]。テスト駆動開発によるセキュリティテストについての可能性については、Tappenden らが HTTPunit を用いて攻撃をテストケースとして作成することで、テスト駆動開発とセキュリティテストの統合を提案している [62]。同様に、Kongsli は受け入れテスト・ツール Selenium を用いたセキュリティ・テストを提案している [46]。

以上から、アジャイルソフトウェア開発にセキュリティ保証を取り組むには、手法の軽量化が重要であることがわかる。また、反復型の開発では、セキュリティの要求や課題を継続して取り扱える仕組みが重要である。セキュリティテスト駆動によるセキュリティ機能の実装もアジャイルソフトウェア開発と整合性が高い。

2.3.4 セキュリティ保証手法とアジャイルソフトウェア開発との整合性の分類

最新のセキュリティ成熟度モデル、BSIMM 第 5 版 (2013) で列挙されているセキュリティ保証の手法について、[19] と同様の分析をおこなった結果を付録 A の表 A.1 – 表 A.12 に示す。BSIMM は比較的大きなソフトウェア開発組織をターゲットとしているため、ソフトウェアのセキュリティを扱う専門部隊、SSG(Software Security Group) や専属の QA(Quality Assessment、品質保証) チームが想定されている。

開発チームの外部にこうした組織の存在が必要となると、アジャイルソフトウェア開発との整合性が失われる。したがって、アジャイルソフトウェア開発の場合は、開発チームへのセキュリティのトレイニングと、開発チーム内で十分にセキュリティ問題に取り組める体制、以上を効率的にサポートできるセキュリティ保証手法（ツール）が必要である。

次に問題となるのが、文書化に依存するセキュリティ手法である。これについては、文書化の負担を低減する手法の導入が必要である。

表 2.8: セキュリティ保証手法とアジャイルソフトウェア開発との整合性 [19]

開発フェーズ	手法	整合性分類			
		整合	自動化	不整合	独立
Requirements	Guideline				独立
	Specification analysis			不整合	
	Review			不整合	
Design	Application of specific architectural approaches				独立
	Use of secure design principles				独立
	Formal validation			不整合	
	Informal validation			不整合	
	internal review	整合			
	external review			不整合	
Implementation	Informal correspondence analysis			不整合	
	Requirements testing		自動化		
	Informal validation			不整合	
	Formal validation			不整合	
	Security testing		自動化		
	Vulnerability and penetration testing		自動化		
	Test depth analysis			不整合	
	Security static analysis		自動化		
	High-level programming languages and tools				独立
	Adherence to implementation standards				独立
	Use of version control and change tracking				独立
	Change authorization			不整合	
	Integration procedures				独立
	Use of product generation tools				独立
	Internal review	整合			
	External review			不整合	
	Security evaluation			不整合	

2.4 モデル駆動開発とセキュリティ保証

この節では、モデル駆動開発とそのセキュリティ保証への適用について述べる。

2.4.1 モデル駆動開発とセキュリティ

2.3 節で述べたように、セキュリティ対策には多大なコストがかかる。そこで、効率的で安全なソフトウェア開発手法として、モデル駆動開発が注目された。特に膨大なコード実装を低減する点と、仕様記述の点でモデル駆動設計は魅力的である。UML と Java のようにモデルからのコード生成が可能であれば、開発の主体をモデリングに移行させる事ができる。この場合、モデルを利用したセキュリティ機能の自動検証が可能になる。UMLsec は、 UML を拡張し、UML 上でセキュリティを扱えるようにするプロファイルの提案である [42] [43]。図 2.5 にモデル駆動型の開発モデルを示す。

モデル駆動開発の問題は、モデルとコードの一貫性の保守である。モデルからコードへの自動生成が利用できない場合、コード上での変更をモデルに反映させなければモデル上でのセキュリティ検証と実態が異なってしまう。また、コードの自動生成が利用できる場合でも、実際に稼働させた状態でセキュリティテストは最終確認として必要である。

モデリングのセキュリティへの活用に関する研究と提案は 2000 年前半に多くなされた。ただし、ソフトウェア開発の対象分野にもよるが、実際の開発においてモデル駆動の手法は普及していない。Web アプリケーションのモデル化についても、モデリングの粒度や視点が異なる様々な手法が提案されている [70] [37]。UML はコード生成機能が実用的ではなく、図による仕様の記述の利用にとどまっている。

また、Web アプリケーション開発が、2.3.2 節に示すようにアジャイルな開発スタイルに移行すると、開発はコード中心の反復開発プロセスとなる。こうした開発では、コードからクラス図や ERD 図の形でモデルの自動生成を行い、モデルを実装結果の理解のために補助的に利用している。

2.4.2 モデル駆動開発のセキュリティに関する関連研究

モデルを利用して、セキュリティテストケースを自動生成する取り組み (MBST: Model-based Security Testing) も研究が進められている [44] [47] [22] 。図 2.6 に MBST を採用した場合の開発プロセスの例を示す。アプリケーションの振る舞いをモデル化することで、網羅的なテストケースの自動生成が期待できる。この手法の課題は、テスト生成可能な正確なモデルの作成である。モデルからのコード生成が利用できない場合、モデル作成とコーディング作業を別途行わなければ

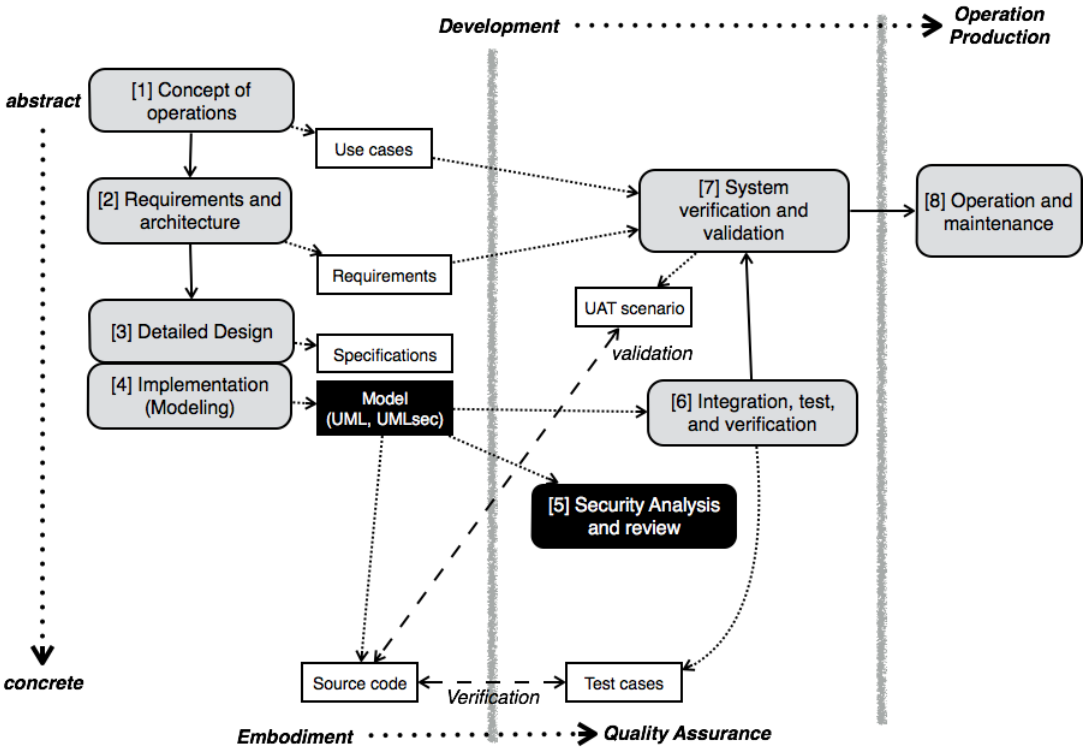

図 2.5: モデル駆動開発の開発フロー

ならない。特にアジャイルソフトウェア開発ではモデルと実装との間に一貫性の問題が発生する。

Xu らは モデル駆動でアクセス制御ポリシーのテストケースを自動生成する手法を提案している [69]。Lebeau らは モデル駆動で Web アプリケーションの脆弱性検査のテストケースを自動生成する手法を提案している [47]。

表 2.9: モデル駆動設計とセキュリティ保証に関する研究

著者 (年)	内容
Jurjens(2004)[42] Jurjens(2005)[43]	UML でセキュリティを扱う UMLsec の提案
Jurjens(2008)[44]	UMLsec + MBST
Halle(2010)[37]	状態遷移図による Navigation Error の検出
Xu(2012)[69]	モデルを用いたアクセス制御のテスト
Lebeau(2013)[47]	MBST、アクセス制御のテスト

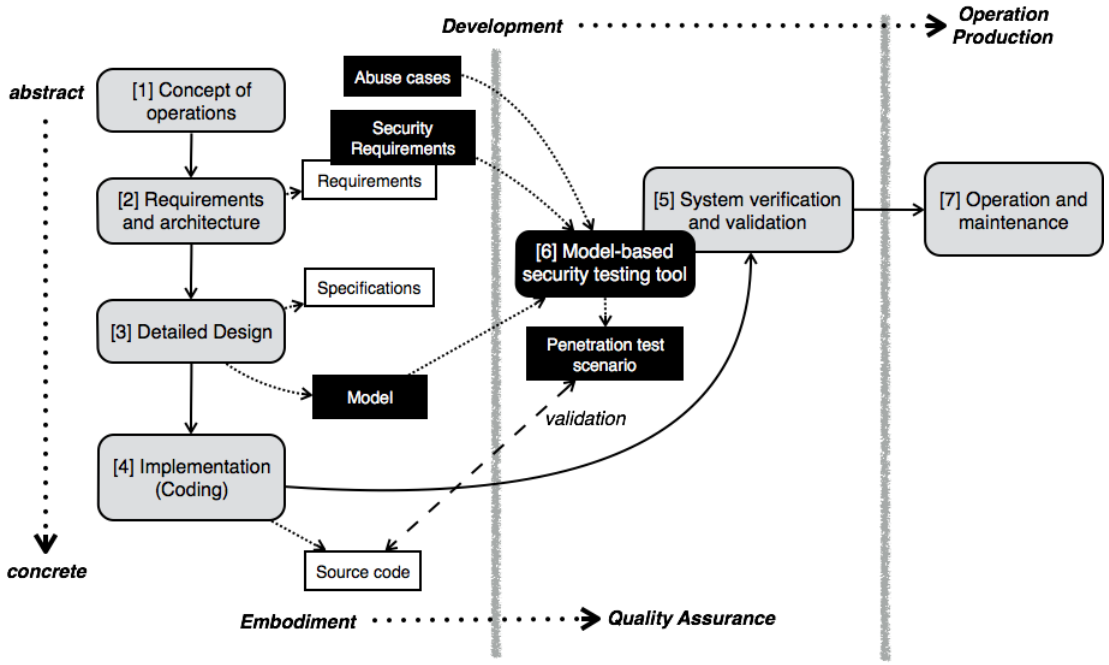

図 2.6: MBST: Model-based Security Testing の開発フロー

2.4.3 Web アプリケーションのモデル化に関する関連研究

Web アプリケーションのモデリングに関しては、Andrews らが状態遷移図を用いて Web アプリケーションの挙動をモデル化する手法、FSMWeb を提案している [12][13]。これは作成した状態遷移図を用いて機能テストケースを生成する事が主な目的である。

モデル駆動開発自体の課題については、 Forward らがサーベイを実施し報告している [35]。モデリングは開発初期には利用されるが、実際の開発においてはコードの開発が主体であることが指摘されている。これはアジャイルソフトウェア開発において、モデル駆動設計が継続して活用できない事を意味している。これは、モデルとコードの一貫性を保つ事が難しく、コードのみが保守されるためである。そして、現在の Web アプリケーション開発においても同様であるといえる。

この課題に答える方法は、1) モデルからコードの自動生成、2) コードからモデルの自動生成である。1) をサポートする開発環境は限られる。特にアジャイルソフトウェア開発で用いるアプリケーション・フレームワークではツールが存在しない。2) については様々な Web アプリケーションのモデル生成の自動化手法が提案されている。Alalfi らは Web アプリケーションのモデル化とテストについて、既存手法を大きく 4 つのタイプに分類している [7]。ナビゲーションモデリングでは、Web アプリケーションの振る舞い (ページ遷移) をモデル化する。初期の Web アプリケーションは静的な構造であったが、現在では動的なページ生成が主流である。相互作用挙動のモデリングでは、Web アプリケーションの (ブラウザ側の) 挙

動をモデル化する。コンテンツモデリングでは、ページの静的な構造をモデル化する。モデル生成はソースコードから行われるものと、実際の動作から行われるもの、双方を利用するものの 3 つのアプローチが存在する。これは、扱う Web アプリケーションの実装方式に依存する。

アジャイルソフトウェア開発で用いるアプリケーション・フレームワークでは、アプリケーション実装のためのコード記述が少ないことが特徴である。つまり、アプリケーションの振る舞いレベルでコードは記述されるため、コードから適切なモデルを生成する手法が有効であると考えられる。

2.5 Ruby on Rails

本節では、本研究の実装評価で用いる Web アプリケーションフレームワーク、Ruby on Rails についてその概要と、セキュリティに関する特性、現在の保証手法について述べる。

Ruby on Rails [11] (以降 Rails と表記) は、スクリプト言語の一種である Ruby 言語で記述する Web アプリケーション・フレームワークである。2004 年に発表され、現在も活発な開発が続いており、表 2.2 で示すように多くのウェブサイトの実装に利用されている。また、Rails を用いた Web アプリケーション開発では、アジャイルソフトウェア開発が実践される。逆に言えば、アジャイルソフトウェア開発が必要なアプリケーション構築で、Rails が Web アプリケーションフレームワークとして広く選ばれている。

2.5.1 Rails の思想

Rails は MVC (Model － View － Controller) アーキテクチャ型の Web アプリケーションフレームワークであり、従来の Web アプリケーション・フレームワークに比べ、大幅に少ないコードの記述で Web アプリケーションの構築が可能である。またフレームワークもアプリケーションも基本的にはすべて、スクリプト言語である Ruby で実装される。Rails は次の 2 つの基本理念に基づいて開発されている。

- CoC: Convention over configuration (設定より規約)
- DRY:Don't Repeat Yourself（重複の排除）

結果として、Rail を用いた Web アプリケーションは非常に少ないコード量で記述することができ、簡単なアプリケーションであれば数分で構築可能である。

2.5.2 Rails の MVC アーキテクチャ

Rails はデータベース駆動型の MVC アーキテクチャを採用した Web アプリケーションフレームワークであり、図 2.7 のように動作する。バックエンドは各種のデータベースで、RDBMS や NoSQL 型のデータベースも利用できる。“Model” は データベースのテーブルを表すクラスである。Rails では ActiveRecord がオブジェクト関係マッピング (ORM) として動作する。この “Model” に対応する形で、“Controller” を配置する。“Controller” の各メソッドに URL が割り当てられ、ブラウザからのリクエストは Rails のフレームワークによってルーティングさ

[11] http://rubyonrails.org/

れ、然るべき ‘Controller” のメソッドが呼び出される。“View” は個々のメソッドに対応する HTML 生成のためのテンプレートである。“View” の記述には ERB[12] や HAML[13] が用いられる。

Rails の基本理念の一つは「設定より規約 (CoC: Convention over Configuration)」である。つまり、Rails の MVC コードは規約に従って配置することで自動的に接続され動作している。このことは、実装コードのモデル変換も容易にしている。

もうひとつの基本理念は「同じことを繰り返さない (DRY:Don’t Repeat Yourself)」であり、関数の形で機能が集約されアプリケーション内で参照されるため、あとで解説するコマンドライブラリによる実装コードの抽象化が機能しやすい。

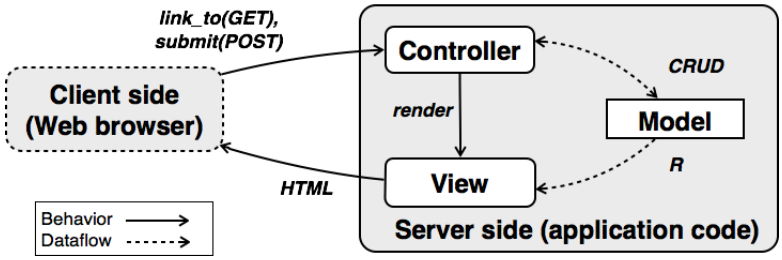

図 2.7: MVC スタイルの Web アプリケーションの振る舞いの概略図

2.5.3 Rails とアジャイルソフトウェア開発

Rails では TDD: Test Driven Development や BDD:Behavior Driven Development を実践する環境が整備されており、実行可能な UAT: User Acceptance Test を起点に開発を進めることができる。又、RSpec[14] や Cucumber[15] といったテスト機能のパッケージを利用することで、テスト駆動開発への対応が容易に実現できる。

その他、Rails は Scaffold と呼ばれるコード生成機能を持ち、MVC コードのテンプレートを自動生成することが可能である。こうしたテストファーストな開発環境により、動くアプリケーションを短いサイクルでリリースしてゆくことも可能となる。

[12] HTML などの文章の中に Ruby スクリプトを埋め込むためのライブラリ, http://www.kuwata-lab.com/erubis/
[13] テンプレートエンジン, http://haml.info/
[14] http://rspec.info/
[15] http://cukes.info/

2.5.4 Railsのセキュリティ機能

XSS、CSRF、SQLインジェクションなどのWebアプリケーション特有の脆弱性については、フレームワークで対応がなされており、セキュリティガイドにしたがって実装する限りにおいては問題は無い。基本的には フレームワークの更新の都度、より “Secure by default” となるように改善が施されている。また、Brakeman[16]などの静的検証ツールを使うことで、実装ミスや、脆弱性のある古いバージョンの使用などの危険はかなり防げる。

アクセス制御などのアプリケーションのデザインに関するセキュリティ機能については、依然として開発者の責任に負う部分が大きく、そうした欠陥が不正アクセスや情報漏洩につながる危険があり、本研究の対象とするところである。

2.5.5 Railsのアクセス制御機能実装

アクセス制御機能はWebアプリケーションの基本機能であり、必要な機能やセキュリティのレベルは、個別のアプリケーションに依存する。そこで、Railsでアクセス制御機能を実装するには次の3つのアプローチがある。

- フレームワークの提供する機能を利用する(HTTPベーシック認証、secure_password)。
- 自前で用意する。
- パッケージを利用する。

複雑な認証の場合は、フレームワークの提供する機能で十分である。一般には、ユーザー管理やロールベースのアクセス制御が必要になる場合が多い。そうした場合は、適切な専用のパッケージの利用が一般的である。表 2.10 に代表的なセキュリティ機能パッケージを示す。認証で広く利用されているパッケージとしては、Devise や Authlogic がある。また、認可では CanCan、TheRole などがあり、ロールベースのアクセス制御 (Role-based Access Control, RBAC) をサポートする。この中で Devise と CanCan が広く利用されている。

一般の開発では、アクセス制御の実装にこうした既存パッケージを活用する例が多い。実際、チュートリアルやサンプルコードを模倣することで、簡単にアプリケーションに組み込むことが可能である。パッケージ化されたセキュリティ機能を使う場合は、パッケージリスト (Gemfile) にパッケージ名を追加しインストールする。その後、必要な設定を行い、コマンドをコード中に記述する。記述方法（コマンドの利用方法）はパッケージにより異なるが、例として、図 2.8 に CanCan の提供するコマンドと実装例を示す。この例に示すように、“Controller” へのコマンドの追加は、A) クラス全体、B) メソッド単位、C) メソッド内のロジックに配

[16] https://github.com/presidentbeef/brakeman

表 2.10: 主要なセキュリティ機能パッケージ

名前	機能	URL
Devise	認証	http://blog.plataformatec.com.br/tag/devise/
Authlogic	認証	http://rdoc.info/github/binarylogic/authlogic
OmniAuth	認証	https://github.com/intridea/omniauth/wiki
Restful-authentication	認証	https://github.com/technoweenie/restful-authentication
CanCan	認可 (RBAC)	https://github.com/ryanb/cancan
Declarative authorization	認可	https://github.com/stffn/declarative_authorization
The Role	認可 (RBAC)	https://github.com/the-teacher/the_role

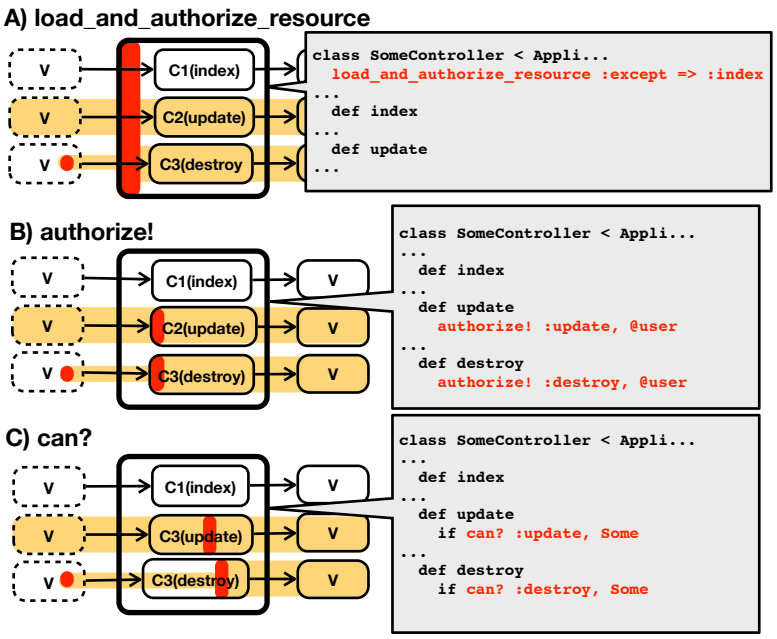

図 2.8: CanCan のコマンド実装方法のバリエーション

置、の 3 種類である。実際のアプリケーションではこれらの様々な実装方法が見られる。この実装の自由度が、Rails におけるアクセス制御機能の実装ミスの原因の一つである。開発者が、利用パッケージのセキュリティ機能を正しく理解していない場合、間違った方法での機能の利用や、間違った設定でアプリケーションに組み込んでしまう危険がある。結果として、アプリケーションに脆弱性が発生する。また、利用している機能の理解が不十分な状態では、セキュリティテストの実施も難しい。

アクセス制御に関する脆弱性（CWE エントリ）とコード上のバグの発生箇所を表 2.11 に 示す。Rails の場合、アクセス制御における PEP(Policy Enforcement Point) は “Controller” に記述するセキュリティ機能を呼びだすコマンドとなる。この配置を忘れると、任意のアクセスや、不適切な権限による Web アプリケーショ

表 2.11: アクセス制御に関する脆弱性 (CWE 定義) と Rails における原因場所との関係

Weakness (and error)	Defect location		
	PDP	PEP	Nav. (View)
Incorrect user management (CWE 286) Improper ownership management (CWE-282)	O		
Improper authentication (CWE-287) Improper authorization (CWE-285) Missing authorization(CWE-862) Incorrect authorization(CWE-863)		O	
Navigation error			O

ンへのアクセスを許す脆弱性となる。PDP(Policy Decision Point) は、利用する認可機能パッケージにより実装方式が異なるが、CanCan の場合は “Model” としてアクセス制御ポリシーがハードコードされる。この記述のミスも、意図しない権限でのアクセスを許す脆弱性となる。一方、“View” テンプレートは、認証や認可に関するアプリケーションの遷移でその権限のチェックが必要である。チェックを忘れた場合には、実際はアクセス出来ないリンクやボタンがブラウザ上に表示され、利用者が選択すると、認証や認可エラーとなる。これ自体は、Web アプリケーション的には脆弱性ではないが、認証認可に伴うバグであり、ナビゲーションエラーと呼ばれる。Rails の開発ではアプリケーションのテンプレートコードの自動生成をよく行うが、その際にアクセス制御に関するアプリケーション構成は考慮されないため、自動生成されたコードを利用者の権限に応じた修正をせずに用いることでナビゲーションエラーが発生する。

図 2.9 は、以上の脆弱性の関係をアプリケーションの制御フローで示したものである。最初の例では、管理権限を持った利用者がブラウザの表示にしたがって管理画面に正規にアクセスする例を示す。2 つ目の例では、一般ユーザーがブラウザの表示にしたがって管理画面にアクセスし、権限が無い旨のエラー表示とともにホームページに遷移させられる、ナビゲーションエラーの典型的な制御フローを示す。3 つ目では “Controller” に権限確認のコマンドの配置を忘れた例を示す。管理者権限での振る舞いテストでは管理画面が表示されるが、実際は管理者以外もアクセス可能な脆弱な状態である。4 つ目の例は 2 と 3 のミスが両方存在する場合で、一般ユーザーが通常の画面アクセスを経て管理画面にアクセスできる。こうした問題は、アクセス制御ポリシーが正しく定義されかつ実装に反映されていない事が原因である。

2.5.6 Rails を使った Web アプリケーションで発生した脆弱性

Ruby on Rails は新しい Web アプリケーションフレームワークであり、アップデートに伴いセキュリティ対策も進んでいる。

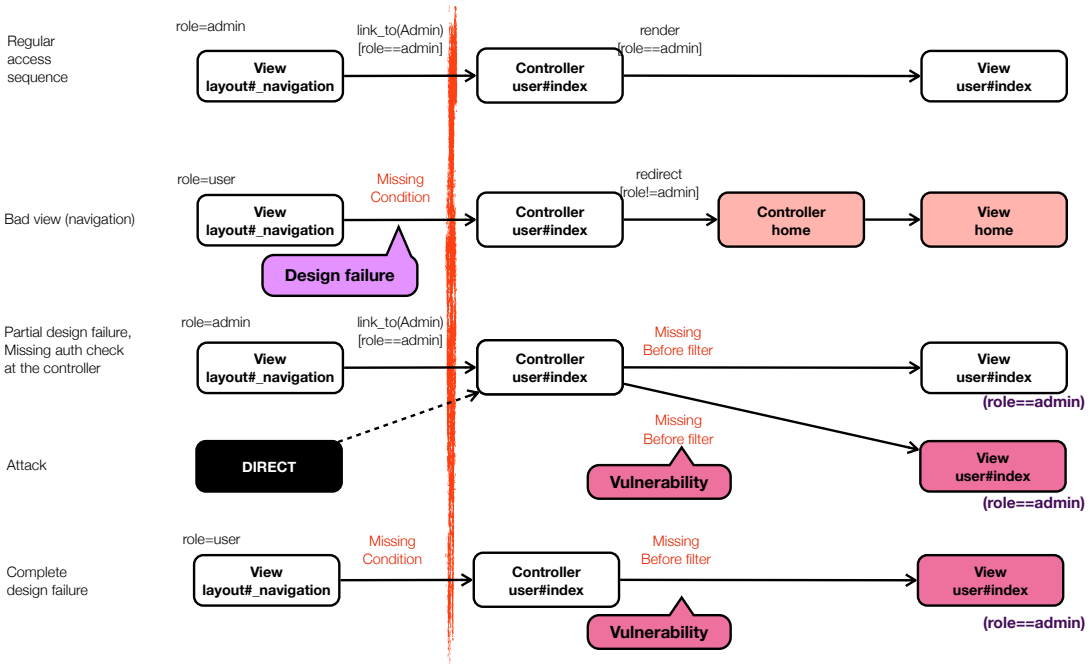

図 2.9: アクセス制御機能の実装エラーパターン

実際に、2006 年から 2013 年までに CVE[17] に登録された Rails を用いた Web アプリケーションで発生した脆弱性について調べた結果が図 2.10、図 2.11 である。図 2.10 は脆弱性がアクセス制御等の設計に関するものか、実装のバグに由来するものかで分類した。図 2.11 は脆弱性がアプリケーションコードのどの部分で発生したものかを分類したものである。Rail の利用が進むにつれて、脆弱性の報告回数が増加している。2011 年以降に CVE のエントリ数が増加している原因の一つは、Rails が商用アプリケーションで利用され始めた結果、ソフトウェアベンダーによる品質保証の導入によって脆弱性が報告されやすくなったことも理由の一つとして考えられる。実際には、Rails を利用した Web アプリケーションは多数存在するが、それがパッケージ化され、CVE に脆弱性が登録されている数は一部であると考えられる。ただし、現在 CVE に登録された脆弱性でその傾向は捉えることができると思われる。

図 2.10 を見ると設計の問題と実装の問題がほぼ同じ数報告されている。設計の問題は、セキュリティ要求が正しく把握されていない事が原因の一つだと推測される。また、脆弱性の発生箇所ではアプリケーションコード部分が多い。したがって、アプリケーションのセキュリティ品質向上が重要であると言える。

[17] Common Vulnerabilities and Exposures, https://cve.mitre.org/

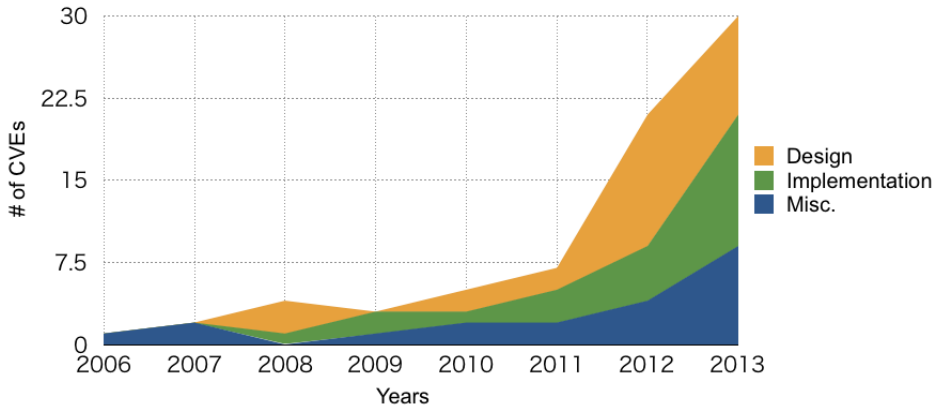

図 2.10: 脆弱性の原因を設計と実装で分類

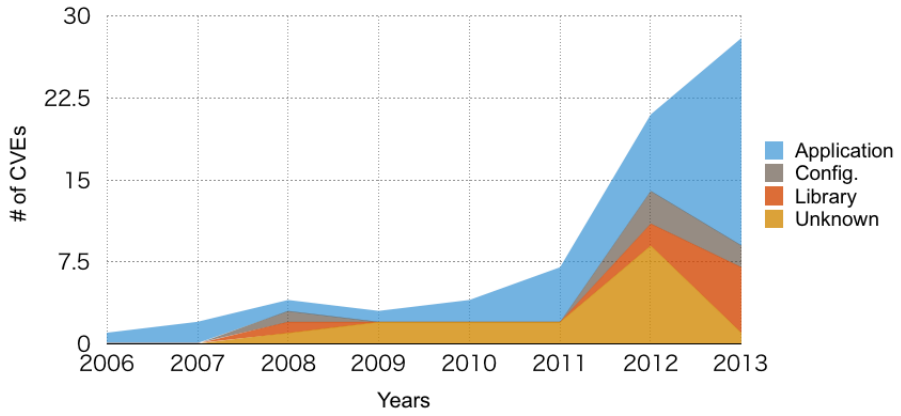

図 2.11: 脆弱性の発生箇所をコード上の位置で分類

2.5.7 Rails とセキュリティ保証手法

Rails では開発者向けにセキュリティガイドを用意している [18]。また、Rails 開発者向けの書籍では、Security on Rails [55] があり、テスト駆動開発を利用したテストケースの作成方法などが紹介されている。ただし、1.2 節でも指摘したが、網羅的なセキュリティテストコードの作成とその保守は容易ではない。

Rails は Web アプリケーションなので、各種の Web アプリケーション脆弱性スキャナによるセキュリティテストが可能である。ただし、脆弱性スキャナの実行時間は (アプリの規模や検査項目にもよるが) かなり長時間となる。そのため、開発作業の中で頻繁に実行し、結果を確認するものではない。また Web アプリケーション脆弱性スキャナで検査可能な問題の多くは、静的セキュリティテストツールでもチェックできるため、アジャイルソフトウェア開発手法におけるコード実装では、静的検証ツールが実用的である。ただし、どちらのツールも認証認可など

[18] http://guides.rubyonrails.org/security.html/

のデザイン的な部分の問題は扱えない。

近年、軽量で実用的な Ruby on Rails 向けの静的なセキュリティテストツールとして、Brakeman[19] や、Codesake-dawn[20] が発表されている。これらは、軽量なコマンドラインツールの形で実装されており、XSS などのポピュラーな脆弱性の危険性や、脆弱なバージョンの利用を警告する事ができるため、利用の拡大が進んでいる [24]。

一方、認証機能など、アプリケーションの設計に関するレベルでの検証に対応する一般的な手法はない。また、Rails のようなフレームワークを用いたコード開発中心の Web アプリケーション開発においては、モデル駆動設計的なアプローチは、モデルの記述作業が負担となり、実際の開発現場では普及していない。特に、Rails のような MVC のアプリケーションフレームワークを用いる場合は、アプリケーションの構造は自明であり、モデルの必要性が低いこともモデル駆動設計が普及しない理由であると考えられる。したがって、2.4 で取り上げたモデル駆動型のセキュリティ保証手法の適用も難しい。

実際に Rails の開発者が利用可能なセキュリティ保証手法をまとめると、1) 開発者はまずセキュリティガイドに従う必要がある。2) 次に、実装したコードの実用的な静的テストツールとして、Brakeman や Codesake-dawn が利用できる。これらのツールは Rails に固有の一般的な脆弱性の検証に有効であり、軽量なため、開発者にも使いやすい。3) Web アプリケーション向けの脆弱性スキャナーは、商用製品も含め、様々なツールが Rails でも利用できる。これらは実際に稼働している Web アプリケーションに対してテストを実施するため、実際の攻撃に対する耐性を自動的に検証することが可能である。問題は、長い実行時間、カバレッジが不明、開発への問題のフィードバックと修正である。

こうした従来のセキュリティ保証手法は単純に実装上の問題を指摘するものが主であり、アクセス制御の検証など、設計に関わるセキュリティの問題の保証をサポートするツールは無い。

2.5.8 Rails のセキュリティ保証に関する関連研究

Rails 向けの静的セキュリティテストの研究では、シンボリック・エグゼキューションを利用したプロトタイプが発表されている [23]。ただし、こうした手法は実験的なレベルで止まっている。原因の一つは、手法の実現が OCAML ベースの処理系にもとづいた非公開のもので、頻繁に更新される Ruby や Rails への対応も行われていない事である。

一方 Laser は、Ruby のみで実装された Ruby 向けの静的検証ツールの実現を試みている [31][32]。開発したツールは公開されており [21] だれでも利用することが

[19] http://brakemanscanner.org/
[20] https://github.com/codesake/codesake-dawn
[21] https://github.com/michaeledgar/laser

できる。そこで、Laser を Rails に適用できないか試みたが、Rails のアプリケーションを解析する場合、フレームワークを含むすべての Ruby コードをパースする必要があるため対象コードが膨大となり、非常に長い解析時間が掛かる事がわかった。また、ツールがサポートしている Ruby の構文が不十分なこともあり、今回試した限りでは Rails への適用は出来なかった。ただし、このツールが Ruby の構文をすべてサポートしたとしても、動的評価部分についての静的検証は困難である。

2.6 アジャイルソフトウェア開発による Web アプリケーション開発とセキュリティ保証の課題のまとめ

Web アプリケーションの開発プロセスが、ウォーターフォール型から、アジャイル型に変化する中で、アジャイルソフトウェア開発と整合性の高いセキュリティ保証の整備が必要とされている。ここでは次の 4 つの課題を取り上げる。

課題 **1** 設計の検証に必要となるセキュリティ要求定義と保守。

課題 **2** テスト駆動開発に適したセキュリティテストの実現による、開発者のセキュリティ保証への関与。

課題 **3** 体系的なセキュリティ保証を実現するための、セキュリティ知識の共有と利用の仕組みの構築。

課題 **4** 様々な変化への迅速な対応。

第 3 章では、ここで指摘した課題を解決する提案手法について詳しく述べる。

2.6.1 課題 1: セキュリティ要求の定義と保守

アジャイルソフトウェア開発とセキュリティ保証手法の各種手法の整合性を見た場合に、まず課題となるのは、文書化や計画性に依存したセキュリティ保証手法である。アジャイルソフトウェア開発では、機能要求の変化に伴い、セキュリティ要求も変化する。この変化を正しく捉え、実装が最新のセキュリティ要求を反映していることを保証する仕組みが必要である。

一般に、アジャイルソフトウェア開発では、文書化よりもコード実装作業を優先する。これはセキュリティの要求定義についても同様である。つまり、文書化されたセキュリティ要求がないか、あったとしても古く、最新の実装を反映していない。

今回の研究を通して Github で公開されている Rails を用いて開発された様々な Web アプリケーションを調査した。その中でセキュリティ要求が明文化され管理された例は存在しなかった。これは公開されているのが実装コードのみであり、そうした文書は非公開であることも考えられる。しかしながら、一般的には、Web アプリケーション開発はある程度成熟しており、ガイドラインや、開発者が持つ暗黙のセキュリティ要求により、実装が進められているのが現状と考えられる。

アジャイルソフトウェア開発プロセスのフローは図 2.4 となる。例えば、テスト駆動開発を進める場合には、そのテストケースが要求から導き出された仕様を表す。しかしながら、セキュリティのような非機能要件を、すべてテストケースの形で仕様化することは現実的ではない。おそらく、セキュリティ要求は開発者の

念頭にあり、コード実装に反映されているはずであるが、それを客観的に検証することは難しい。セキュリティ保証を実施するためには、最新の実装に対応するセキュリティ要求を明確に定義（文書化）する必要がある。

また、開発の過程で、セキュリティ要求の変更が必要となり、その変更をテストケースや実装コードに反映させる必要が生じるケースも想定されるが、セキュリティ要求が暗黙のまま変更されると、それに伴うセキュリティ実装の変更を検証する手立てが無い。

このように、アジャイルソフトウェア開発におけるセキュリティの一つの大きな課題は不明確なセキュリティ要求である。結果として、実装が対応すべきセキュリティ機能を十分には検証できない事である。開発プロセスの柔軟性とセキュリティ保証の十分性が相反する要素となっている。

2.6.2 課題2: テスト駆動開発に合致した網羅的なセキュリティテスト

つぎに、アジャイルソフトウェア開発で、能動的にセキュリティ機能の実装を扱うには、セキュリティテスト駆動開発が重要である。その際に、テスト駆動開発の利点を阻害しない戦略的な手法が必要となる。具体的にはセキュリティテストの網羅性を確保しつつ、開発者が扱える量のテストケースの自動生成が必要である。

セキュリティテストとは、特定の脆弱性に対する攻撃と、その対応を確認することである。ただし、脆弱性スキャナでは、すべての箇所に可能性のある全ての攻撃を想定するため、組み合わせが膨大となり、実行に非常に長い時間がかかるため、アジャイルソフトウェア開発において開発者が日常的に用いるセキュリティチェックのツールにはならない。

効率的なテストケースを自動生成するためには、場所の特定と、その場所で考慮すべき脆弱性の限定が必要である。これには、モデル駆動開発をベースとしたセキュリティ保証の様々な取り組みが有望であると考える。ただし、要求、モデル、実装の過程が独立しているとアジャイルソフトウェア開発では適用が難しい。

2.6.3 課題3: セキュリティに関する情報の共有

セキュリティ要求が文書化されないことも、セキュリティに関する情報の共有の課題の一つである。実際のセキュリティ保証では様々な判断を開発者は求められている。例えば、セキュリティ要求やセキュリティ機能選択の妥当性や有効性、テスト・ツールの出す警告への対応などである。これらは非常に手間のかかる作業であるため、知識として開発チームで共有し、有効に再利用することが、アジャイルソフトウェア開発でセキュリティ保証を効率的に進めるうえで重要である。2.1 節でとりあえげたセキュリティ知識を開発者が自然に活用できる仕組みが望ましい。

2.6.4 課題 4: 変化への迅速な対応

アジャイルソフトウェア開発では、様々な変化への対応が求められている。これにはセキュリティ要求の変化も含まれる。

アジャイルソフトウェア開発では、短い開発サイクルで顧客の要望をくみとりながら、差分的に機能を追加、変更してゆく。セキュリティ保証では、要求の変化、実装の変化に伴うセキュリティの問題を的確に検出し、修正方法をナビゲートする仕組みが必要である。具体的には、セキュリティ保証は開発者が使いやすいツールの形で実現される必要がある。動作速度、扱いやすさ、実装作業との整合性の高さが実用上重要である。

第3章　提案手法

アプリケーションフレームワークを利用することで、開発者はアプリケーションを少ないコード量で実装できる、また、フレームワークがアプリケーションのテスト環境も提供することで、開発者はテスト駆動開発を実践することができる。こうしたフレームワークや開発環境の利用がアジャイルなソフトウェア開発を可能にしている。開発者は自動生成されたアプリケーションコード、サンプルコードや類似機能を持つアプリケーションの実装を参考にして、迅速にアプリケーションの実装を進める。しかしながら、こうした実装作業において、アプリケーションのセキュリティに関して充分な注意が払われていないために、脆弱性を含むコードが生まれる危険もある。

実装コード中の脆弱性を発見する手法としては、静的解析ツール（2.2.3.2 節）の利用が有効である。静的解析ツールのメリットは、ツールの実行速度が速いこと、実装コードレベルで問題を指摘することなどが挙げられる。したがって、こうしたツールを開発者自身がコード実装作業と併用して利用することで、迅速な脆弱性の修正が可能となる。一方で、検知できるパターンや精度には限界がある。特にスクリプト言語を用いた Web アプリケーションフレームワークでは、全ての実装が単一のスクリプト言語で記述されるため、フレームワーク部分とアプリケーションの実装部分との境界が曖昧である。そのためフレームワーク部分の大量の実装コードを解析が静的検証の精度を高めるために必要となり、ツールの実行速度が課題となる。また、アクセス制御機能を実現するコマンドが正しく配置されていることの確認のように、アプリケーションの持つセキュリティ機能の実装上の問題には対応できない。

本研究では、静的解析ツールのように開発者が開発の中で頻繁に利用可能なツールの形で、セキュリティの問題をより包括的に取り扱える手法の実現を目指した。そこで注目したのが、フレームワークが提供する機能（API, コマンド）のセキュリティ上の特性である。一般に、フレームワークが提供する API のドキュメントには、その機能（入出力、振る舞い）についての解説はあるが、そのセキュリティについて体系的な記述がない。そこで、セキュリティ特性をコマンドのレベルで分類し活用することで、様々なセキュリティ保証の仕組みが自動化できると考えた。また、フレームワーク部分のセキュリティ特性を事前に定義し、静的検証をアプリケーションの実装部分に限定することで、ツールの実行速度の大幅な改善が期待できる。

本章では、本論文で提案するセキュリティ保証手法の詳細を述べる。まず、3.1 節でアプリケーションの開発作業を 6 つに分類し、それぞれと提案手法との関係についてその概要を述べるとともに、2.6 で指摘した課題との対応を整理する。次に、提案の中心となるコマンドレベルの抽象化と、それを用いたモデル生成について 3.2 節で説明する。3.3 節ではセキュリティテストのテストカバレッジの計測手法について、3.4 節ではコマンドレベル抽象化ライブラリを用いたセキュリティ知識との自動リンクについて説明する。最後に 3.5 節で提案手法のアジャイルソフトウェア開発への組み込みについてまとめる。提案手法の実装と評価については次章で扱う。

3.1 概要

本研究では、コード中心の開発において、開発者がセキュリティ要求やセキュリティテストの実施をコード開発と平行して実行できる仕組みを実現する。その中心となるのがコマンド抽象化ライブラリ（Command Abstraction Library, CALib）である。ここで言う「コマンド」とはアプリケーションフレームワークが提供する API(Application Programing Interface) を指す。このライブラリに、フレームワークが提供するコマンド (API) の振る舞いとそのセキュリティ属性を分類して記録することで、効率的なアプリケーションの振る舞いの抽象化（モデル化）とそれを用いた各種のセキュリティ保証を実現する。具体的には、ソースコードの静的検証、セキュリティ要求の抽出と実装との整合性確認、セキュリティテストカバレッジの計測、セキュリティ知識と連携した脅威分析、などの実施がこのライブラリを利用することで可能となる。

図 3.1 にアプリケーション開発と提案手法のセキュリティ保証との関係を示す。提案手法では、アプリケーションのソースコードがすべてセキュリティ保証の起点となる。開発者はコーディング作業を中心にテスト駆動 (図の上部、Application development 部分) でアプリケーションの開発をすすめながら、セキュリティ保証の作業を開発の中に取り込んでゆく。セキュリティ保証で必要となるアプリケーションフレームワークや、ソフトウェア開発に関するセキュリティ知識は、コマンド抽象化ライブラリ (Command Abstraction Linrary) の形で提供される。提案手法を実装したツールが、この知識を元にアプリケーションコードからセキュリティ上の問題点を自動的に洗い出す。

このツールの動作を簡単に説明する。

1. **dynamic test** テスト駆動開発 (TDD) でアプリケーションを開発する。その際に、セキュリティ機能のテストも記述し、セキュリティ機能の実装をすすめる。

2. **parse** 実装されたアプリケーションコードをコマンド抽象化ライブラリを参照

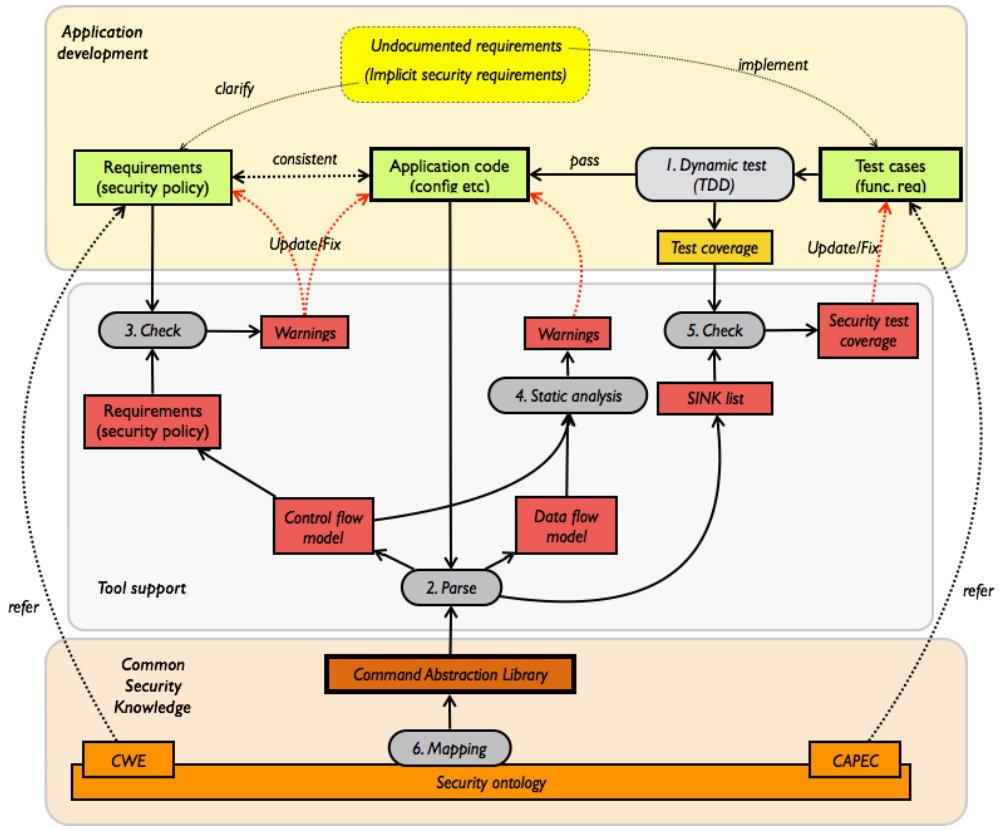

図 3.1: アプリケーションの開発 (上部) とツールによるセキュリティ保証のサポート (中部) とセキュリティ知識 (下部) との関係

しながら解析し、アプリケーションを制御フローモデル (Control flow model) とデータフローモデル (Data flow model) に変換する。

3. **check** 制御フローとコード上のセキュリティ機能から「コード由来のセキュリティ要求」を抽出し、開発者が定義した「開発者定義のセキュリティ要求」と照合することで、要求と実装の一貫性を確認する。不一致があった場合は、要求もしくは実装を修正する。

4. **static analysis** モデルを用いて静的検証を行う。 問題があった場合はアプリケーションコードを修正する。

5. **check** (2) のパースの際に、コード上のセキュリティ的に確認が必要な部分を「SINK」として抽出し、テストカバレッジとの照合から、セキュリティ・テストカバレッジを求める。これの結果を参考に TDD で用いるテストケースにセキュリティテストを埋め込んでゆく。

6. **Mapping** アプリケーションのセキュリティ要求や、テストケース作成には、コマンド抽象化ライブラリを介してセキュリティ知識を参照できるようにする。

図中のグレーの角丸長方形はツールによる処理を示す。(1) のテストの実施はフレームワークのテスト機能であるが、(2)〜(6) は提案手法のツールで実現する。(2)〜(5) については、コードやテストケースの変更の都度実行する。(6) についてはセキュリティ要求やテスト手法の確認の際に実行するため、その使用頻度は (2)〜(5) に比べると低い。このように単一のフレームワークでセキュリティ要求から実装、テストまでのセキュリティ保証に対応できる点が従来にない新しい仕組みである。

2.6 で列挙した課題との対応でみると、課題 1 のセキュリティ要求の定義と保守は、要求定義と実装の一貫性を確認する、左側のループが対応する。また、課題 2 のテスト駆動開発へのセキュリティテストの組み込みは、テストカバレッジを計測する右側のループが対応する。課題 3 のセキュリティ知識の共有は、コマンド抽象化ライブラリによるフレームワークの提供するコマンドのセキュリティ特性を抽象化することで実現している。課題 4 の変化への迅速な対応は、図で示すように要求、実装 (のコード)、テストの一貫性を保証する仕組みを提供することで、要求、実装、テストそれぞれを起点とした変更がセキュリティに与える影響を捉えることで対応する。

次に、本手法で提案する各セキュリティ保証についてその概要を説明する。3.1.1 節は、セキュリティ要求に基づくアプリケーションのセキュリティ実装の静的検査、3.1.2 節は、セキュリティテストカバレッジ計測によるセキュリティテストケース作成の支援、3.1.3 節は、セキュリティ知識と実装との対応によるセキュリティ要求の確認とセキュリティテスト作成の補助、最後に 3.1.4 節でコマンド抽象化ライブラリの概要を示す。

3.1.1 明確なセキュリティ要求の作成と、実装との一貫性の確認 (静的テストによるセキュリティ保証)

2.6.1 節で課題とした「セキュリティ要求の不備」はアジャイルソフトウェア開発におけるセキュリティ要求管理の共通の問題である。提案手法では、(存在するであろう) セキュリティ要求は結果としてセキュリティ機能として実装コードに現れていると仮定する。これを実装コードから捕捉し、セキュリティ要求を明示的に定義し保守するための仕組みを実現する。

アプリケーションがサポートしているセキュリティ機能は、それを実現するコマンドを特定することで推定することができる。そのため、提案手法では「コマンド抽象化ライブラリ」で個々のコマンドが関連するセキュリティ機能を指定する。図 3.1 の 2. Parse で、アプリケーションのソースコードを解析から、アプリケーションの振る舞いモデルを生成する際に、モデル上の各状態にそれぞれのセキュリティ機能の有無を記録してゆく。この情報を整理することで、コード由来のセキュリティ要求を作成する。例えば、アクセス制御の場合には、図 1.1 の例

で示したように、認証認可のコマンドの有無が、その状態（Controller）の認証認可の必要性の要求の有無となる。

一方、開発者も開発者定義のセキュリティ要求を作成し、コード由来のセキュリティ要求と比較することで、要求と実装との整合性を確認する。初期の開発者定義のセキュリティ要求の作成は、コード由来のセキュリティ要求を複製することで簡単に生成できる。開発者は生成されたセキュリティ要求をレビューし、意図と異なる箇所を修正し、不明な部分は補足する。これは開発初期の反復で実施し、コードレポジトリに一緒にコミットする。このセキュリティ要求がベースとなり、以降の反復に引き継がれる。

一旦、セキュリティ要求が作成されれば、そのセキュリティ要求を元に実装の静的なセキュリティテストが可能となる。反復開発でコードが変更する場合、小さなコード変更や要求変更によるセキュリティ要求と実装の不整合は容易にチェックできる。

重要なのは、セキュリティ要求の文書化、保守、そして共有であり、提案手法はコード実装が開発の中心となるアジャイルソフトウェア開発において、そのための新しいフレームワークを提供する点である。

提案するセキュリティ保証のフレームワークでは、実装コードをパースしてアプリケーションの「振る舞いモデル」を生成する。したがって、モデル生成の際に、静的なセキュリティ検証の実施が可能である（これは一般的な静的セキュリティテストツールと同様の機能である）。提案手法のツール化では、3.2.3 節で解説するコマンド抽象化ライブラリを用いて、脆弱性の危険が想定される箇所を警告としてレポートする。

3.1.2 セキュリティテストのカバレッジ計測 (動的テストによるセキュリティ保証)

静的なテストでは、セキュリティ機能の配置が適切であるかのチェックは可能であるが、セキュリティ機能が実環境で正しく動作することは保証できない。したがって、セキュリティ機能が実際に期待した動作をすることは、動的テストを用いて実環境で挙動を確認する必要がある。また、静的検証で問題が指摘されたが実際には問題がない場合などの確認も、動的テストで脆弱性が無いことを確認することができる。

テスト駆動開発においても、適切なセキュリティテストケースを作成することで、セキュリティ機能の実装を進めることができる。また、静的検証で問題が指摘された箇所が False-Positive (擬陽性) であることの確認も実施可能である。

ただし、網羅的なセキュリティテストを開発者が作成することは生産的ではなく、セキュリティテストケースのカバレッジが計測できれば開発者は計画的にテストケースの作成と保守が可能になる。そこで、提案手法では、セキュリティに

関するコマンドをコード中での配置にもとづいたテストカバレッジの計測を行う。また、制御フローモデル、データフローモデルによるテストケース作成の支援も可能である。

本手法ではアプリケーションが利用しているすべてのセキュリティ機能と配置を把握しているため、効率的なテストケースの選定が可能である。つまり、静的テストによるセキュリティ機能の配置の十分性をチェックし、動的テストによる個別のセキュリティ機能の動作確認を実施することで、セキュリティテストの網羅性を保証することができる。

3.1.3 セキュリティ知識との連携

開発者が十分なセキュリティ知識を持っていない場合、実装コードにどのようなセキュリティ上の問題が存在するか把握できない。また、セキュリティに関する知識を持っている場合でも、最新の状況で網羅的にセキュリティ問題を把握することは非常に属人的である。提案手法では、コマンド抽象化ライブラリを用いて、セキュリティに関係するコマンドを 2.1 節で説明した CWE、CAPEC と連携することで、体系的なセキュリティ知識の中でアプリケーションのセキュリティ機能の確認とセキュリティテストケースの整備を支援する。

3.1.4 コマンド抽象化ライブラリ

以上のセキュリティ保証の仕組みの共通のコアとなる部分が「コマンド抽象化ライブラリ」である。コマンド抽象化ライブラリではアプリケーションが参照するコマンドをすべて網羅する。ただし、実際に定義を行うのはモデル生成とセキュリティに関するコマンドのみである。モデル生成に関しては、制御フロー、データフロー生成に関するコマンドを分類する。セキュリティに関しては、セキュリティ機能を実現するコマンド（Security Command、SC）とセキュリティ上の問題を引き起こす可能性のあるコマンド（Risky Command、RC）の 2 種類に分類する。また、コマンドに関連する、CWE,CAPEC の情報（番号）も付加する。

3.1.5 課題への対応のまとめ

提案手法が、どういう方針で 2.6 節で挙げた 4 つの課題を解決するかを簡単にまとめる。

課題 **1** 設計の検証に必要となるセキュリティ要求定義と保守については、セキュリティ要求定義と実装コードの一貫性の自動チェックにより、要求やコード実装の変更に伴う問題の発生を開発者が把握できるようにすることで解決する。

課題**2** テスト駆動開発に適したセキュリティテストの実現による、開発者のセキュリティ保証への関与については、セキュリティテストのカバレッジの自動計測により、開発者が必要十分な数のセキュリティテストケースをテスト駆動開発の作業の中で直接配置できるようにすることで解決する。

課題**3** 体系的なセキュリティ保証を実現するための、セキュリティ知識の共有と利用の仕組みの構築については、コマンド抽象化ライブラリにおける、フレームワークのセキュリティ機能のライブラリ化と、CWE,CAPEC のセキュリティ知識を実装コードレベルで参照できる仕組みの導入とで解決する。

課題**4** 様々な変化への迅速な対応については、上記の自動化の任意のタイミングでの実施で対応する。軽量なツールとして実装することで、開発者は手軽に提案手法を用いた検証作業を実行できる。

3.2 セキュリティ機能のモデル表現

この節ではセキュリティ機能のモデル化について述べる。ここで言うセキュリティ機能とは、アクセス制御のように開発者がコード実装するアプリケーションのセキュリティ要求や設計に関するものと、クロスサイトスクリプティング（XSS）のようなWebアプリケーション固有の脆弱性に対する対策である。提案手法ではWebアプリケーションを制御フローとデータフローの２つを用いてモデル化する。本手法ではこの２つのモデルのセットを「セキュリティ検証モデル」と称する。

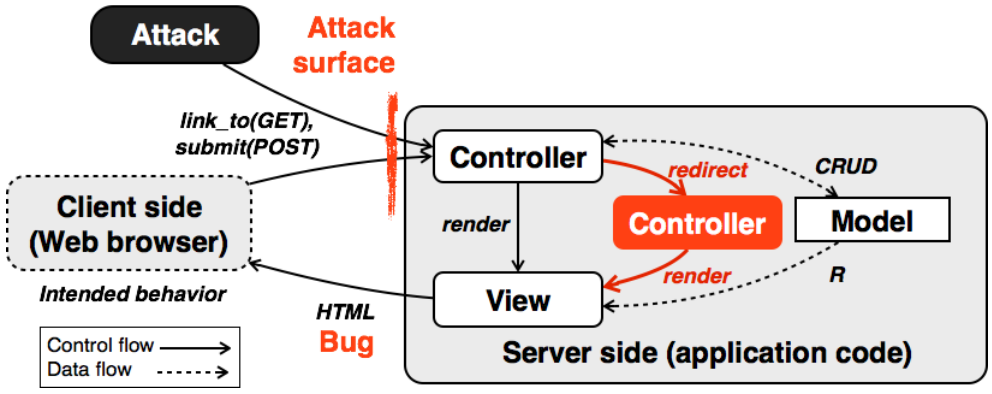

図 3.2: MVC スタイルの Web アプリケーションの状態遷移モデル概要

3.2.1 制御フローモデル (Web アプリケーションの振る舞いの状態遷移による表現)

セキュリティ機能をモデル化する場合、アプリケーションへの攻撃と、それに対するアプリケーションの挙動（セキュリティ対策）が、モデル上に表現される必要がある。したがって、攻撃の起点となる外部インターフェースを中心とした状態遷移モデルがこうした振る舞いの表現に有効である。一般に攻撃に晒されるインターフェースをアタック・サーフェースと呼ぶ [49]。Model-View-Controller (MVC）スタイルの Web アプリケーションの振る舞いのモデル化の例を図 3.2 に示す。ここではサーバーへの入力リクエスト (HTML メッセージ) を主な Web アプリケーションのアタック・サーフェースとする。入力メッセージは Controller により処理され、必要に応じて Model が更新される。その後、View によりページ (HTML) が動的に生成され、ブラウザにリクエストに対するレスポンスとして送られる。通常はブラウザで表示されるページ情報に従い、次のリクエストがサーバーに送られるが、アプリケーションに対する攻撃の場合は、任意のリクエストがサーバー側に送られる。 したがって、想定された入力しか受け付けないように Controller は実装されなければならない。不正な入力があった場合には、外部か

らのリクエストは別のエラー処理の Controller にリダイレクトされ、Web アプリケーションはエラーメッセージ等のページに遷移する。このように、MVC スタイルの Web アプリケーションは Controller と View を状態 (state) とした状態遷移モデルで表現することが可能である。このモデルで、正常な動作と攻撃とその対応双方をモデル化することが可能である。

提案手法では、MVC スタイルの Web アプリケーションの動的な振る舞いを、Controller、View を状態とした 状態遷移モデル (これを制御フローモデル、またはナビゲーションモデルとよぶ) をアプリケーションのソースコードから自動生成する。

3.2.2 データフローモデル（外部との情報の授受の表現）

提案手法では、ブラウザとの入出力、Controller、View と Model の間のデータのやり取りをデータフローモデルとして表現し、制御フローモデル同様にアプリケーションのソースコードから自動生成する。

コード自体の実行フローとしては、 Controller から Model への遷移は存在する。また View はページ生成に際して Model 上のデータを参照する。Controller と Model と View は基本的に対で存在するため、制御フローモデルとしては Controller と View を用いる。Model への外部からのデータの授受については データフロー モデルを構築することで対応する。アプリケーションで保持する変数についても外部との授受はデータフローモデルで取り扱う。データフローの捕捉は、SQL インジェクションや、クロスサイトスクリプティング (XSS) の解析に必要になる。また、アプリケーションが管理するデータへのアクセス制御の解析においても必要となる。

3.2.3 コマンド抽象化ライブラリ (アプリケーション・フレームワークの抽象化とモデル生成の高速化)

2.4 節で指摘したように、モデル駆動設計の課題はモデルの作成と保守である。とくにコードの自動生成が利用できない場合は、実装コードとモデルの乖離が起こるため、モデルは開発初期の文書化ツールの域を出ない。

提案手法では、セキュリティ検証モデルをアプリケーションの実装コードから生成 (リバース・エンジニアリング) する手法を取る。これはコーディング中心のアジャイルソフトウェア開発において、モデルを活用するのに適した手法である。

例えば、Rails のような Web アプリケーションからモデル生成する際の課題は、フレームワークを含めてアプリケーションすべての実装が Ruby で行われており、解析対象となる全コードは膨大である。これらのコードをすべてパースし、モデ

ルを生成することは現実的ではない。また、Ruby は動的型付け言語なため、静的なソースコードのパースでモデルを生成することが難しいケースも多く存在する。

この問題に対処するために、提案手法では解析するソースコードはアプリケーションの実装コードに限定し、フレームワークや外部コンポーネントによって提供される機能を別途ライブラリ化して利用する。具体的には、アプリケーションコード内で参照しているすべてのコマンド (API) を「コマンド抽象化ライブラリ」に登録する。その際に、モデル生成で必要となる特性、セキュリティ上の特性をコマンドの属性として記録する。

コマンド抽象化ライブラリを用いたコードからのモデル生成フローの概要を図 3.3 に示す。ツールはアプリケーションコード部分 (MVC の実装部分) を一旦 AST(Abstract Syntax Tree) に変換、AST をパースしながらモデルに変換する。その際にコード上に現れたコマンドは、コマンド抽象化ライブラリを参照して、制御フロー、データフローを抽出する。また、コマンド抽象化ライブラリに記録されたコマンドのセキュリティ属性を、生成したモデルの上の属性として付加する。

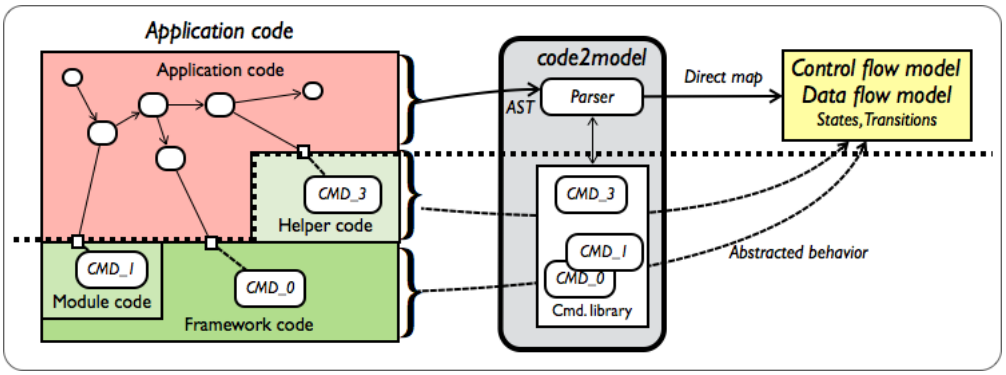

図 3.3: コマンド抽象化ライブラリを用いたコードからのモデル生成フロー

コマンド抽象化ライブラリとアプリケーション開発との対応を図 3.4 に示す。コマンド抽象化ライブラリはその対象に応じて複数存在する。ツールとしてサポートするコマンド抽象化ライブラリは、アプリケーション・フレームワークと標準的なサードパーティの提供するパッケージである。アプリケーション固有のコマンドについては、アプリケーションで独自にコマンド抽象化ライブラリを作成する。例えば、図 3.3 のヘルパーやモデルで実装されたコマンドはそれぞれのアプリケーションで定義する。Rails の場合、テストケースも Ruby のコードで実装されるため、テストケースの解析にもコマンド抽象化ライブラリを用いることができる。アプリケーション側の機能として新たに実装されたコマンドは、初期段階ではライブラリに存在しないため、不明なコマンドとしてエラーとなる。開発者はモデル生成上重要なコマンドからこれらの定義を順次ライブラリに追加してゆく。

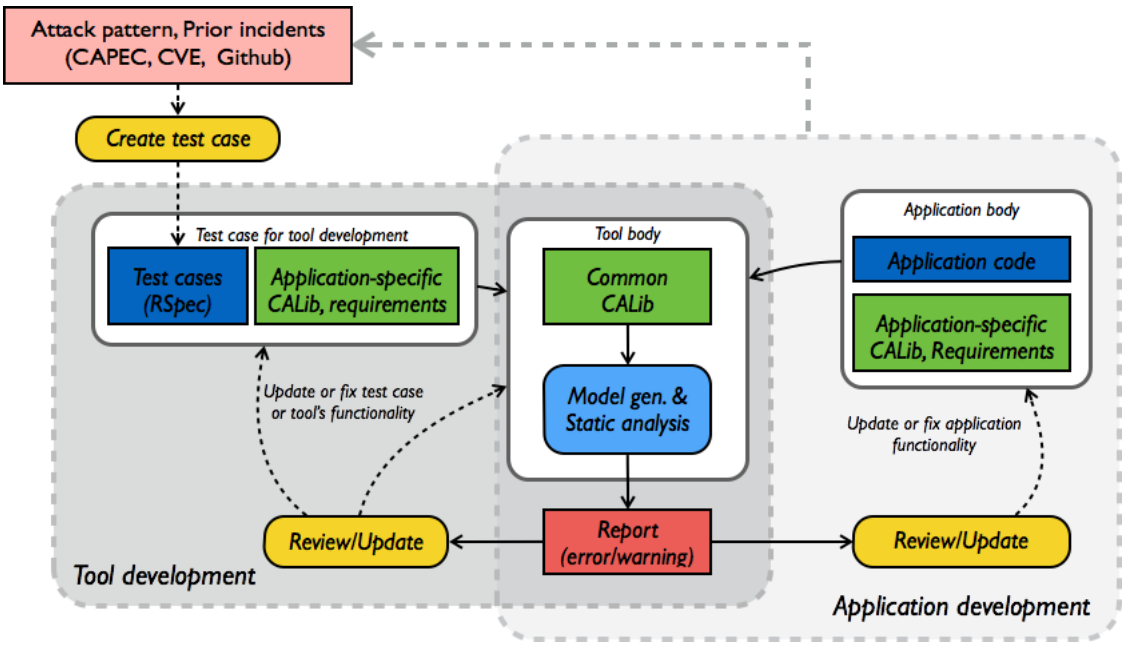

図 3.4: コマンド抽象化ライブラリのとアプリケーション開発との対応

3.2.4 セキュリティポリシーの記述と検証

セキュリティ検証モデルにはコード由来のセキュリティ属性が記録される。2.5.5 節で Rails のアクセス制御実装の特徴を示したが、こうした認証や認可のコマンドの配置から、モデル上の個々の状態のセキュリティ属性を抽出する。これがコード由来のセキュリティポリシーとなる。

一方、セキュリティ機能の実装の正しさを確認するためには、明確に定義されたセキュリティ要求定義が必要である。これを開発者定義のセキュリティポリシーとする。このポリシー作成や保守は開発者が実施する必要がある。そのため、開発者の負担を低減するために、記述はできるだけシンプルに行える事が望ましい。

提案手法では、MVC タイプのアプリケーションの構造に着目し、まず各 Model に対してそのセキュリティポリシーを設定し、その Model を関連する Controller に伝搬、次に Controller に対応する View に要求設定を伝搬する手法を採用した。例外として、認証に関する Controller などでは、その前後でセキュリティ状態が変化する (未認証の状態から認証された状態に変化する)。こうした場合には、Controller によって Model での設定とは異なるセキュリティ要求が必要な場合がある。そのため、個別の Controller レベルでもセキュリティ要求を設定できるようにする。

次に、Controller と View の関係だが、通常は Controller とその対応する View のセキュリティ要求は共通である。しかしながら、Rails の例では複数の View が Form という形で 別の View を共有する場合がある。この場合は、Form を呼び出

す View (及び View を呼び出した Controller) のセキュリティ要求が共通であれば問題無いが、異なる場合は注意が必要である。例えば、アプリケーションユーザーのロールが異なると振る舞いが変わる。こうした場合に対応するため、Form View では 複数のセキュリティ要求を受け入れられるようにし、その伝播元の View を記録することで、実際の状態遷移の際に親の View のセキュリティ要求を参照できるようにした。またデータフローモデル上でもポリシーを比較できるようにするため、Model のセキュリティ要求を Model の持つ個別の Attribute（モデル内の変数） に伝搬させておく。

開発者が定義するセキュリティポリシーの設定フローを図 3.5 に示す。ポリシー記述は、Model 単位で記述する Base Policy と、Controller や View の状態のポリシーを個別に設定する、 Exceptional Policy がある。定義したポリシーは、Model から、Controller、 View に伝搬させ、すべての状態でポリシーを明確に定義する。

セキュリティ検証モデル上で、コード由来のセキュリティポリシーと開発者定義のセキュリティポリシーを比較することで、セキュリティ要求と実装の一貫性を確認する。以上は PEP に関するコマンド配置の検証である。PDP についてはアクセスコントロールテーブルを作成し、そのレビューで妥当性を確認する必要がある。

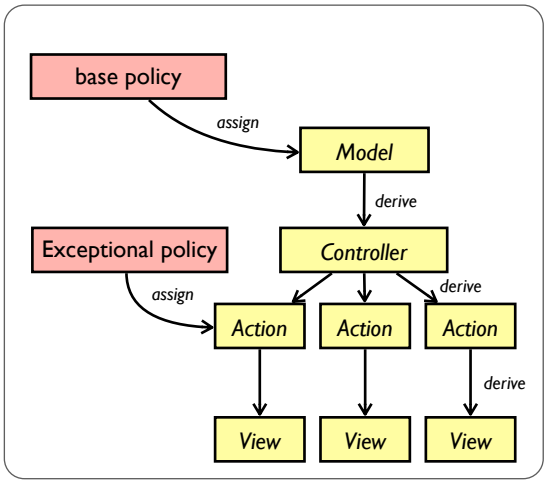

図 3.5: モデルへの開発者定義ポリシーの注入

3.2.5 セキュリティコマンドの分類と静的テスト

セキュリティに関するコマンドは、Security Command (SC) と Risky Command (RC) に分類する。SC にはアクセス制御などのセキュリティ機能を実現するコマンドを登録する。アクセス制御に関する SC については 3.2.4 節で説明したよ

うに、別途定義するセキュリティ要求との一貫性を検証する。RC にはその使用が脆弱性の原因となるコマンドを登録する。

セキュリティ検証モデルと SC,RC との関係には、図 3.6 に示すようにデータフローに関して主に 6 つのパターンがある。オレンジの箱が外部からの入力であり、緑の箱は外部の出力しても安全な内部変数、もしくはサニタイズ済みの外部入力である。XSS(stored) の例では、Web ブラウザからの外部入力 (HTML_IN) はフレームワークにより内部配列 (params[]) に格納されアプリケーションの Controller に渡される。Controller では入力されたデータを変数 (X2) に代入にされ、Model でデータベースに保存される。Model に保存された変数 (X2) は View で参照され、生成された HTML メッセージ (HTML_OUT) は Web ブラウザに出力される。この際に、変数 (X2) はフレームワーク内で “h” 関数を用いてエスケープ処理され、安全な文字列として HTML メッセージ (HTML_OUT) に組み込まれる。このように、Rails の場合は、View で出力されるアプリケーションの内部変数はすべて (“h” コマンドで)HTML 的に安全な文字列にエスケープされる。しかしながら “Bypass” のように “raw” コマンドを用いると、フレームワークでのエスケープ処理が無効化される。これはアプリケーション内部で生成した HTML を含む文字列を出力する際に用いられる。この例では、出力されるデータは安全な内部のコンスタント変数のため問題はないが、外部入力がこのデータパスに含まれる場合は、XSS の危険がある。そのため “raw” コマンドは RC に分類される。同様に、“find” コマンドは SQL インジェクションの危険が、“eval” コマンドはコマンドインジェクションの危険がある。こうした RC コマンドに、外部入力から直接入力がないかはセキュリティ検証モデルのデータフローで Taint Propagation 解析を行うことで検知できる。ただし、RC への外部変数の入力が 途中 “sanitiza_sql” コマンドや “check” コマンド、“path_check” コマンドなどで安全性が保証されていれば問題はない。これらのコマンドは SC として登録する。

セキュリティ検証モデルを用いた静的テストでは、アプリケーション全体のデータフローを追えるため、通常の静的テストツールよりは広いスコープで検査を実施できる。しかしながら、検知能力と精度はモデル化の精度に依存する。そこで、次節で述べる動的テストと組み合わせることで、より確実なセキュリティ保証を実現する。

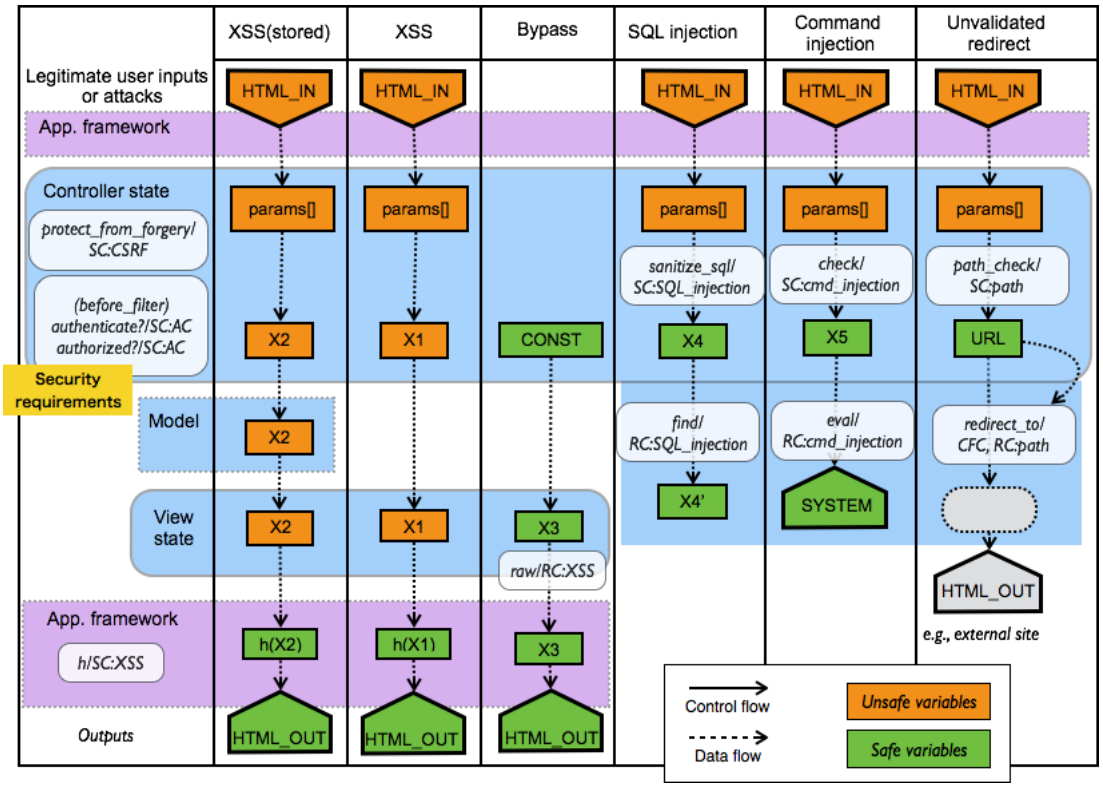

図 3.6: セキュリティ検証モデルと SC,RC との関係

3.3 セキュリティテストカバレッジの計測

この節では、コマンド抽象化ライブラリを用いたセキュリティテストのカバレッジの自動計測手法について説明する。コマンド抽象化ライブラリを用いてソースコードを検査することで、ソースコード上に配置されたセキュリティ機能 (SC) や危険な箇所（RC）を特定することができる。この箇所を「SINK」と呼ぶ。このSINK のコード上の位置情報とテストケース実行によるテストカバレッジを突き合わせることで、既存のテストケースがどの程度セキュリティテストを含むか計測することができる。SINK を用いた脆弱性の発生箇所の特定とセキュリティテストのカバレッジ計測は Dao らによって提案 [26][27] されているが、ここでは、このアプローチをセキュリティ機能にも拡張し、SC,RC を含んだアプリケーション全体でのセキュリティテストのテストカバレッジに利用する。

テストケースと SC,RC の関係を図 3.7 に示す。テストコード中の cmd1 は入力データのセットを行うコマンド、cmd2 は指定したページへのアクセスのコマンドである (この際、HTML の POST メッセージが生成され、アプリケーションに送られる)。このメッセージに対するアプリケーション内部の制御フローは、まず Controller state の、PEP を実現する cmd3(SC) により、権限が確認される。確認で問題がなければ、cmd4 により View state に状態遷移し、出力メッセージが生成される。データフローについては、入力されたデータは Model の cmd5(SC) で安全に処理され、View では　cmd6(RC) でフレームワークでのエスケープ処理を無効化されて、出力メッセージに埋め込まれる。

ここで、PEP を実現する cmd3(SC) の有無については、その状態（Web ページ）に正規と不正規の権限でアクセスすることでテスト (動作の確認) が可能である。一方 XSS の場合、入力から、cmd5(SC)、cmd6(RC) を経て出力されるデータフローに XSS となるメッセージを入力することで、cmd5(SC) が正しく機能しており、cmd6(RC) の使用が問題ないことを確認する。こうしたテストケースを SC,RC それぞれに作成し、実行する。テスト実施後のテストカバレッジに、該当する SC,RC のコマンド実行が含まれることを確認する。

3.3.1 SINK による RC カバレッジの計測

提案手法で用いる SINK とは、一般に外部入力データが入力されるセキュリティ上注意が必要な箇所である。データフロー解析により、外部入力が XSS や SQL インジェクション攻撃となる箇所を特定し、そこを SINK と呼ぶ。テストでは SINK に対して、攻撃データをインジェクションし、問題がないことを確認する。UAT(User Acceptabce Test) レベルのテストケース記述を行うには、データフローと制御フローの双方の情報が必要となる。

手順は以下のようになる。

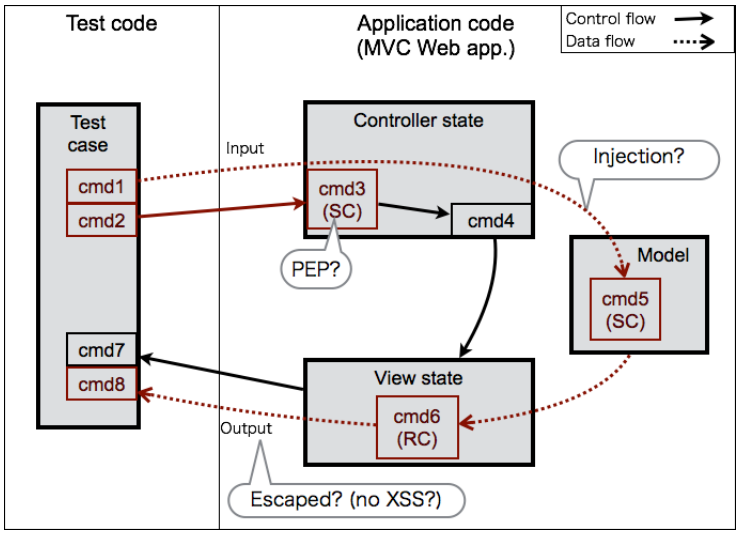

図 3.7: セキュリティテストとカバレッジの計測

1. コマンド抽象化ライブラリに Risky Command (RC) を定義する。
2. ソースコードを検査し、RC コマンドの存在箇所を SINK として登録する。
3. RC の使用が潜在的な脆弱性をもたないことを実際のテストケースで確認する。
4. テスト実行後のテストカバレッジ (ステートメントカバレッジ) を用いて、SINK の箇所がテストされていることを確認する。

脆弱性の種類によってテストケースの記述方法は異なる。一般には、不正な外部入力が問題を起こさないことを確認する。テストケース作成については、次節で述べる CAPEC の情報などを参照し、プログラマが作成する。RC 由来の SINK については、基本的にはソースコード上のすべての SINK がテスト対象となる。つまりセキュリティテストカバレッジの母数は RC の出現数であり、分子はそのテストの有無となる。

3.3.2 SINK による SC カバレッジの計測

一方、特定のセキュリティ機能 SC の場合は、SC が期待した振る舞いをすることと、SC の配置が適切であることの 2 つの確認が必要である。

SC の機能確認については、ソースコード上の存在する SC すべてをテストする必要はない。SC の正しい配置はセキュリティ要求で定義される。つまり、セキュリティ要求を定義しないと配置の妥当性が検証ができない。これについては 3.2.4 節で説明した静的検証で保証できる。

SC の提供するセキュリティ機能の確認については、SC 単体でのユニットテストの実施と、サンプリングでコード中の任意の SC を選択し、その SC に関する機能テストの実施を行う。したがって、SC の SINK テストカバレッジの母数はコード上で利用されている SC の種類であり、分子はそれぞれのテストの有無となる。

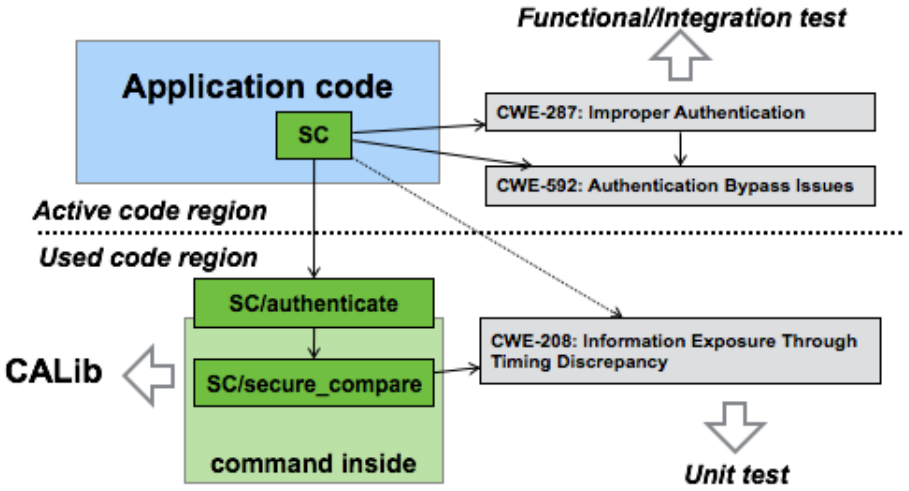

図 3.8: 単体機能テストと結合テストによる SC の確認

認証コマンドと、認証コマンドの内部実装とテストの関係の例を図 3.8 に示す。アプリケーションコード中の SC に関しては、CWE-287(認証の不備) や CWE-592(認証のバイパス) のような脆弱性が無いことを確認するためのテストを、機能テストや結合テストレベルで記述する。一方、SC コマンド単体についてはユニットテストで機能を確認する。この例では、CWE-208(データ比較の時間差に起因する脆弱性) の脆弱性対策が施されていることを確認する。

アクセス制御のテストについて、セキュリティコマンド (SC) の配置とテストの関係を図 3.9 にまとめる。テストケースに参照となるアクセスコントロールリスト (ACL) を持つ場合、これがアプリケーションの PDP のルールとして用いられる。また、テストケース中の ACL をアクセス制御ポリシーに変換し、テストケースを記述する。

3.3.3 開発プロセスへの埋め込み

提案手法で得られるセキュリティテストカバレッジを用いることで、テスト駆動開発へのセキュリティテストの組み込みを計画的に実施できるようになる。アジャイルソフトウェア開発フローと提案ツールを利用した セキュリティ保証の関係を図 3.10 に示す。アプリケーションコードは一旦セキュリティ検証モデル (Application model) に変換 (Parser 1, P1) され、テストカバレッジの計測 (Parser 2, P2) を行う。このメトリックスをフィードバックとして、テストケースの更新や、セキュリ

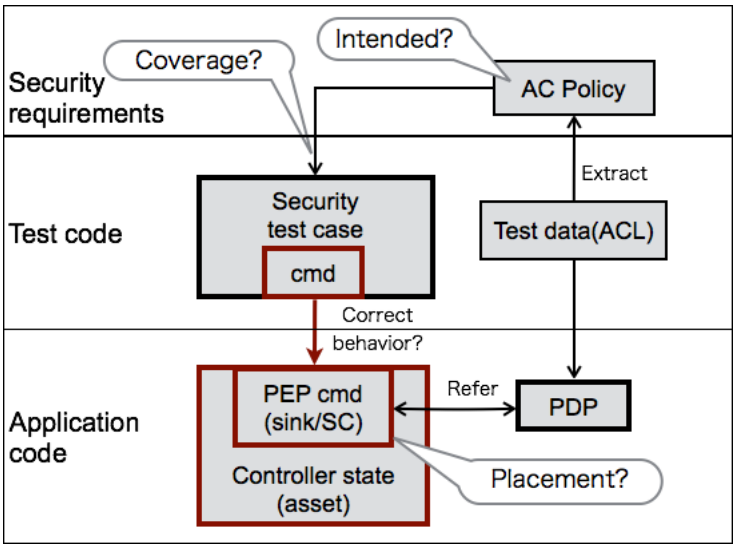

図 3.9: アクセス制御機能のテスト

ティ要求定義の更新を行う。個別の問題の理解と修正にはトレース機能を用いて、特定の警告について詳細を調べる。トレースモードでツールを実行すると、関連するモデルの情報の詳細がツールの実行時に表示される。一連のツールの動作の詳細については 4.1.5 節、図 4.4 で詳しく説明する。

テスト駆動開発では機能要求がまずテストケースとして記述され、テストをパスするようにアプリケーションコードを開発する。認証のような一部のセキュリティ機能はこの流れで開発することができるが、この段階ではまだセキュリティテストとしての網羅性は担保していない。そこで、本手法で計測されるテストカバレッジを元に、必要なテストケースを追加する。

SC についてはその配置については要求定義と静的検証で対応し、実際の機能については、その機能を実装しているアプリケーション上の任意の箇所をサンプリングしてテストケースを作成する。RC については、基本的には RC を使用しているすべて箇所について、テストケースを作成する。これらについての実際の数の評価については次章で検証する。

網羅的なセキュリティテスト実施のために、大量のセキュリティテストケースを作成することは現実的ではない。提案手法は、アプリケーションの内部知識を元に、必要最小限のセキュリティテストケース作成を支援する。

ここで計測されるメトリックスと開発者の対応、優先度を表 3.1 にまとめる。最初の作業は、コマンド抽象化ライブラリの不備や、遷移が不明な制御フローモデルの修正を行い、セキュリティ検証モデルの精度を高める事である。次に、アクセス制御に関して、アセットとロールのテストを整備する。SC、RC の SINK カバレッジについては、ツールが指摘するプライオリティにしたがって、テストを整備する。

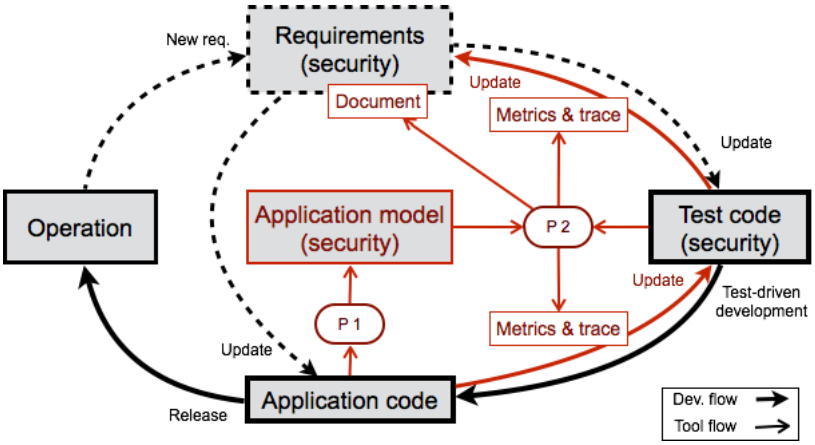

図 3.10: アジャイルソフトウェア開発フローと提案ツールを利用した セキュリティ保証の関係

表 3.1: 計測されるメトリックスと開発者の対応

Metrics	action	priority
Num. of command	review	low
Num. of command (unknown)	add definition	high
Num. of command (tentative)	add definition	mid
Num. of trans.	review	low
Num. of trans. (unknown)	add definition	high
Num. of trans. (tentative)	add definition	mid
Asset cov. (AA_F)	update TC or tool	high
Role cov. (AR_F)	update TC or tool	high
Sink cov (SC_{Unit})	update TC or tool	mid
Sink cov (SC_{Func})	update TC or tool	mid
Sink cov ($RC_{Func-high}$))	update TC or tool	high
Sink cov ($RC_{Func-mid}$)	update TC or tool	mid
Sink cov ($RC_{Func-low}$)	update TC or tool	low

3.4 セキュリティ知識と実装との連携

セキュリティ要求定義とセキュリティ知識の共有が、アジャイルソフトウェア開発におけるセキュリティ保証の課題であると 2.6 節で指摘した。セキュリティ専門家ではない一般の開発者が、アプリケーション開発（コーディング）を行いながら実装と関連するセキュリティの知識を参照できるようにすることが、この問題に対する解決策の一つである。そして、コマンド抽象化ライブラリはこの解決に利用できることをこの節で示す。コマンド抽象化ライブラリを介して、開発者が普段コードに記述するコマンド (API) にセキュリティ知識をリンクさせる。セキュリティ知識として、2.1.2 節で解説した CWE、CAPEC を用いる。これらを用いるメリットは、脆弱性や攻撃パターンが番号で一意に特定できるため、コマンド抽象化ライブラリへの登録が簡単な点である。また、CWE は最も保守されている脆弱性に関する知識であり、同様の物は他に存在しない。

セキュリティ知識は多岐にわたる。CWE はソフトウェアにおける様々な脆弱性を網羅するが、セキュリティ保証対象に関連する脆弱性はその一部である。そこで、CWE で列挙される脆弱性を該当アプリケーションに関連するものに絞る。次に、セキュリティに関係するコマンドを CWE と対応付けることで、コマンド抽象化ライブラリを経由して実装コードとセキュリティ知識を関連付ける。

一例としてクロスサイトスクリプティング (XSS) に関係する Rails のコマンドと CWE,CAPEC の関係を図 3.11 に示す。CWE-79(XSS) の対策 (countermeasure) が “h” コマンドであり、その無効化つまり、XSS を引き起こす (triger) 可能性のあるのが “raw” コマンドである。また、XSS の攻撃パターンは CAPEC-18 に定義されている。コマンド抽象化ライブラリでは、それぞれのコマンドに CWE-79 と CAPEC-18 を関連付ける。ただし、この図で示すように SC と RC で意味が異なる。

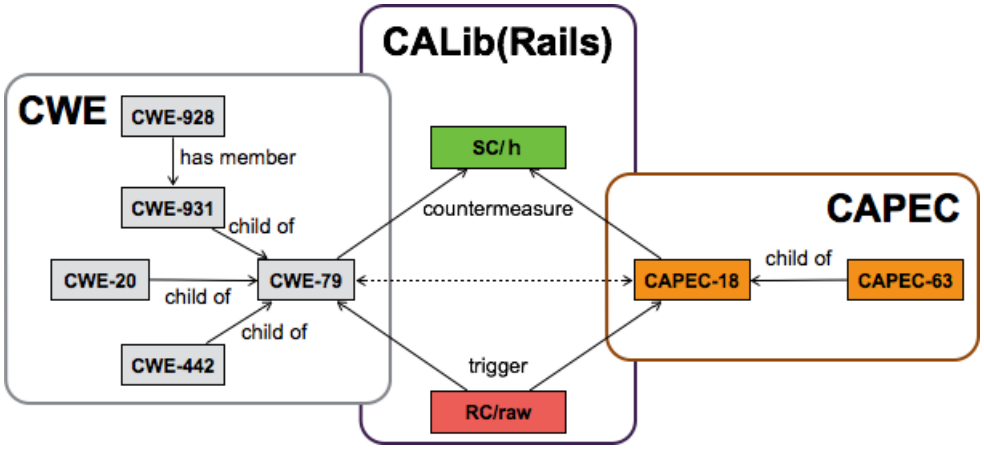

図 3.11: CWE、CAPEC とコマンド抽象化ライブラリの関係の一例

引き続き 3.4.1 節でセキュリティ知識の選択について説明する。続いてに 3.4.2

節でセキュリティ知識とセキュリティ要求、セキュリティテストの関係について整理する。

3.4.1 セキュリティ知識の選択

CWE、CAPEC はソフトウェアの問題すべてを網羅しており、特定のアプリケーション用としては利用しづらい。そこで、開発するアプリケーションに関連のある脆弱性情報となるようにセキュリティ知識の選択を行う。具体的には、CWE のグラフ構造を利用して、CWE の脆弱性定義を特定のアプリケーションドメインに選択する。これは次の 3 つのステップで実施される。

ステップ **1,** 準備： CWE の定義をグラフで扱えるように変換する。CWE は XML で公開されており、その依存関係を任意のツールで容易に木構造のグラフとして扱うことができる。

ステップ **2,** アプリケーションドメイン選択： 最初のステップでは対象アプリケーションに対応する上位の定義を選択し、その下位（枝）の定義をすべて選択する。実際には CWE の View タイプを選択することになる。例えば、Web アプリケーションの場合は OWASP の Top10 (CWE-928) や Web Problem (CWE − 442) などである。

ステップ **3,** アプリケーションフレームワーク選択： これは対象フレームワーク以外の問題の選択を解除する。例えば対象が Ruby on Rails であれば、Java や PHP 固有の脆弱性は関係ない。

以上の作業で CWE 定義の中から開発中のアプリケーションに関連する脆弱性のみで構成された CWE のサブセットグラフが得られる。実際には、現在の CWE 定義では不十分なために、上記の分類では選択されない場合がある。例えば、Rails における Mass Assignment (CWE-915) が該当する。こうしたエントリは個別に追加することで対応する。

CAPEC についても同様に、CAPEC の XML 定義からグラフを作成する。CWE と CAPEC の関連は CWE,CAPEC ですでに定義済みであるため、CWE のサブグラフと連携させるかたちで、CAPEC のサブグラフを得ることができる。

セキュリティ知識（CWE,CAPEC）の選択と、コマンド抽象化ライブラリ (CALib)、実装コード、テストケースの関係を図 3.12 に示す。CWE、CAPEC はドメイン選択と、アプリケーションフレームワーク選択で、対象のアプリケーションに該当するサブセットとなる。ソースコード上のコマンドはコマンド抽象化ライブラリでの定義にしたがって、CWE,CAPEC と関連付けられる。ここでのメトリックス（図ではカバレッジ) は、CWE のサブセットグラフに関連するコマンド数と、CAPEC と関連するテストケース数である。このメトリックスはあくまで指標であり、想

定される脆弱性を網羅することで、開発者が意識していなかった問題を発見することが目的である。アプリケーションにセキュリティ上の問題がないのであればCWE や CAPEC との関連は小さくなる。以上の手法を実際のアプリケーションに適用した結果の詳細は次章で示す。

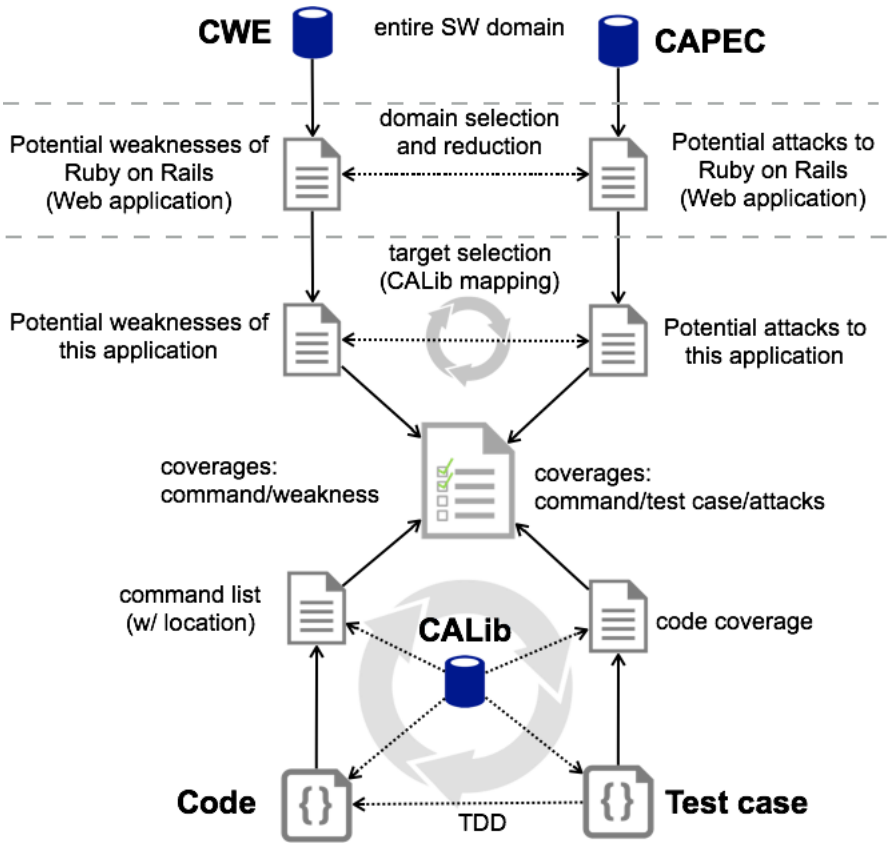

図 3.12: CWE、CAPEC の項目選択と CALib によるアプリケーションコードとテストケースとの関連付けの関係

3.4.2 セキュリティ知識を用いたセキュリティ保証

セキュリティ知識を用いることで、コマンド抽象化ライブラリでのセキュリティ関連コマンドの体系的な分類が可能となる。図 3.13 は図 3.1 のセキュリティ知識と開発プロセスとの関係をより詳細に示したものである。アプリケーションコードを解析することで CWE、CAPEC とセキュリティ関連コマンドとの関係が体系的なグラフの形で得られる。選択された CWE を元に、セキュリティ要求の妥当性を確認することができる。また想定しているにもかかわらず選択されなかった CWE がある場合は、コマンド抽象化ライブラリの定義が不十分である可能性がある。セキュリティテストケース作成は CAPEC を参照して行う。

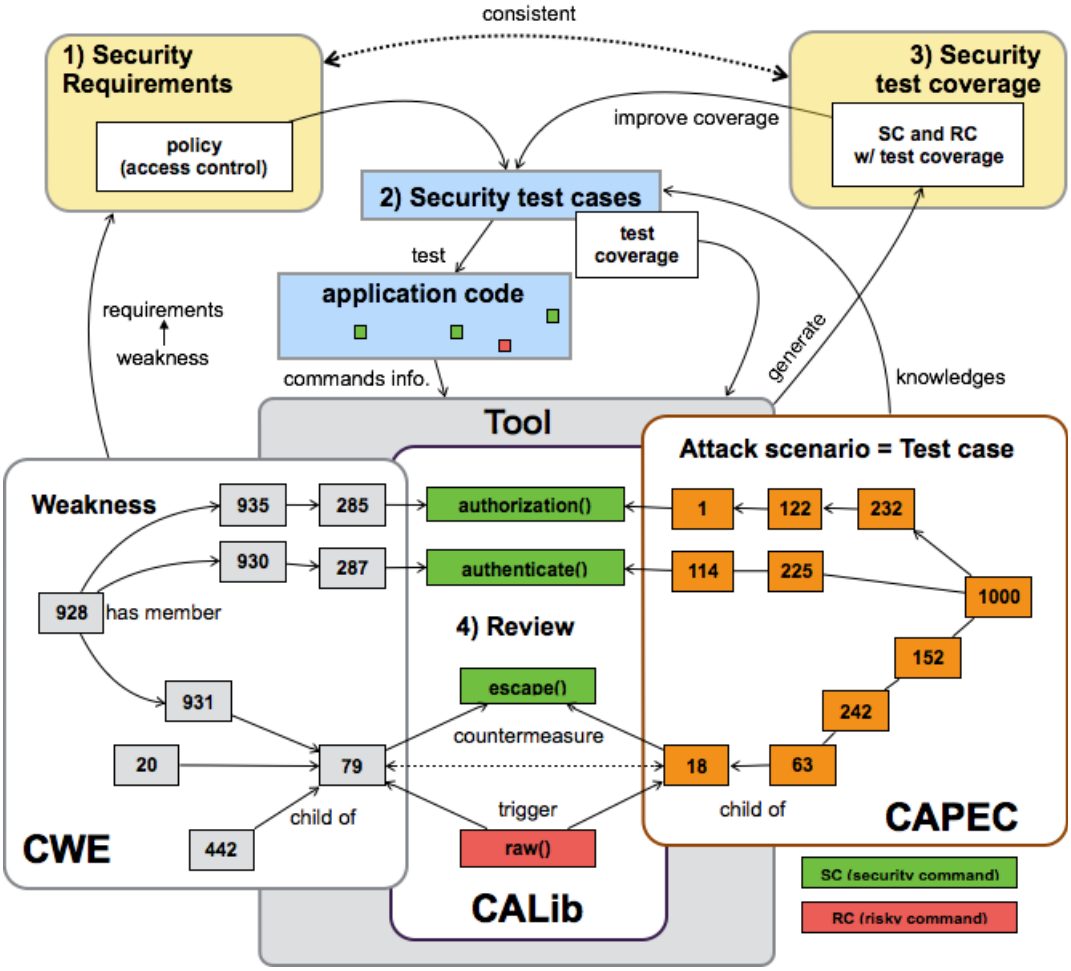

図 3.13: セキュリティ知識によるセキュリティ要求定義とセキュリティテストカバレッジの計測

3.5 アジャイルソフトウェア開発への組み込み

この節では提案手法のアジャイルソフトウェア開発への組み込みについて整理する。図 2.4 で示したアジャイルソフトウェア開発プロセスに提案手法をマップしたものが図 3.14 である。ツールを介してアプリケーションの持つセキュリティ要求、機能、テストを関連付けることで、コーディング中心の開発プロセスと平行する形でセキュリティ保証を実現する。

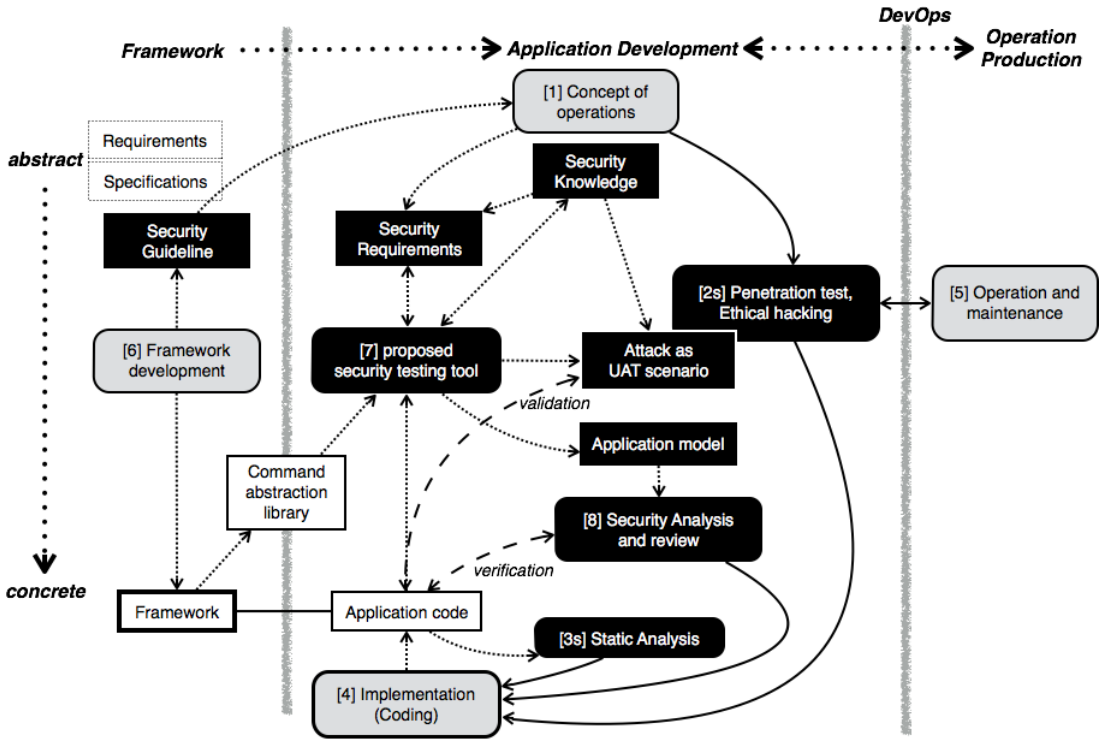

図 3.14: Agile 開発モデルと提案手法

提案手法を開発者が用いる際に、コマンド抽象化ライブラリの整備や、セキュリティ要求の作成などの作業が必要となる。これらを最初にすべて整備することは初期段階の開発コストを高くする。そこで、ツールの実行では、問題があった場合その修正方法を提示することで開発者の作業を補佐する。また、ツール化にあたり、セキュリティに関する情報を一覧できるダッシュボードを提供する。ダッシュボードでは提案手法以外のセキュリティ保証ツール (主に静的検証ツール) と連携できるようにする。

3.5.1 改善対策を提示するインターフェース

モデルの自動生成や、要求定義、テストケース生成などの作業をすべて自動化することは難しい。まず、要求定義はアプリケーションのステークホルダーや開発

者の意図で決めるべきものである。また複雑なロジックを含むアプリケーションコードのモデル化には限界があり、状態遷移が追えない場合がある。テストケース生成についても UAT の実装パターンにバリエーションが多く、完璧な自動生成は難しい。そこで、不明な点を洗い出し、対話的なインターフェースを通して開発者が情報を補完することで、ツール自体の複雑化を避けるようにする。対象となるフレームワークやアプリケーションも絶えず変化しており、本ツール自体の保守性の確保も重要である。

3.5.2 ダッシュボードによる情報の共有

セキュリティ要求、実装、テストを一つのツールで一元管理するため、対象とする Web アプリケーションのセキュリティ保証をダッシュボードの形でまとめることができる。ツールがレポートした警告、モデルの状態、テストカバレッジを表の形で提示する。結果として、開発に携わるすべてのステークホルダーがアプリケーションのセキュリティ保証の現状を確認することができる。また、アプリケーションの機能拡張時には、機能拡張に専念し、その後、本ツールを用いてセキュリティ実装の更新を行うことで、開発者に必要とされてきたセキュリティ対策への負担の低減も実現できる。

3.5.3 外部ツールによる機能補完

提案手法は、主にアプリケーションの実装コードに関係するセキュリティの問題に対応するが、実稼働するシステムレベルですべてのセキュリティ問題に対応できるわけではない。そこで、実用的な外部のセキュリティテストツールと連携することで、アジャイルソフトウェア開発におけるセキュリティ保証をより包括的に実施できるようになる。

静的検証ツールはその警告を本ツールのモデルにマップすることで、その検証用のテストケース生成が可能となる。また、Web アプリケーションスキャナについては、本フレームワークによって管理されているアクセス制御ポリシーを利用して、適切な認証情報や設定を Web アプリケーションスキャナに与えることで、Web アプリケーションスキャナの実施をサポートすることが可能である。

最終的には、ダッシュボードにすべての検証結果が一覧表示され、セキュリティ保証の現状把握が容易となる。

具体的な例は 4.1.7 節で詳しく解説する。

3.6 まとめ

この章ではアプリケーション・フレームワークのセキュリティに関する情報をコマンド抽象化ライブラリに集約して、実装コードからアプリケーションの振る舞いモデルを生成することで、セキュリティ要求の抽出、静的セキュリティ解析、動的セキュリティテストカバレッジ の計測に活用する手法を示した。次の 4 章ではアジャイル型開発を代表する Web アプリケーションフレームワーク、Ruby on Rails を対象とした セキュリティツールの開発と実験評価の詳細を示す。

第4章　提案手法のツール化と実験

3 章で提案したセキュリティ保証の手法を、実際に Rails に対して適用するために開発したツールが、この章で解説する RailroadMap である。本ツールは、スクリプト言語、Ruby、で記述することで開発環境との親和性を実現する。パッケージの形で開発環境に簡単に追加する事ができ、また、必要に応じて自由に開発者が修正や拡張を加える事ができる。実行はその他の開発ツール同様にコマンドラインから行う。

提案手法の実用化（ツール化）にあたり、次の 3 つの目標を設定 した。1) 要求や設計に起因するセキュリティの問題と、実装に起因するセキュリティの問題とを統一した手法 (ツール) で取り扱えること。2) コマンド抽象化ライブラリの作成保守が大きな負担にならないこと。3) アプリケーション付随の回帰テストでセキュリティ保証を行える (テスト駆動開発にセキュリティ テストが組み込める) ことである。

本章では、4.1 節で RailroadMap の開発、設計、機能について説明する。次に実際の Rails アプリケーションに対して本ツールを適用することで次の 4 つの実験を行いツールの機能を確認した。4.2 節ではセキュリティ要求と実装の一貫性の確認と静的テスト (機能評価 1)、4.3 節ではセキュリティテストケースの生成 (機能評価 2)、4.4 節ではテストケースカバレッジの計測 (機能評価 3)、4.5 節ではセキュリティ知識との連携である (機能評価 4)。さらに、4.6 節では、既存のセキュリティテスト・ツールとの機能比較を行った。2.6 節で指摘したアジャイルソフトウェア開発におけるセキュリティ保証の 4 つの課題と、ツール化における目標と評価実験との関係を表 4.1 に示す。

表 4.1: 目標及び評価実験と 4 つの課題との関係

課題	目標 1	目標 2	目標 3	機能評価 1	機能評価 2	機能評価 3	機能評価 4
1. セキュリティ要求の定義と保守	✓	✓		✓			✓
2. TDD に合致したセキュリティテスト	✓	✓	✓		✓	✓	
3. セキュリティに関する情報の共有		✓					✓
4. 変化への迅速な対応		✓		✓		✓	✓

4.1 RailroadMapの開発と機能

4.1.1 RailroadMapの開発方針と機能

提案手法を実現化するにあたり、アジャイルソフトウェア開発の観点から、本ツールに求められる実装上の目標は、次の３つであると考える。

目標**1** 要求や設計に起因するセキュリティの問題と、実装に起因するセキュリティの問題とを統一した手法で取り扱えること。

目標**2** コマンド抽象化ライブラリの作成保守が大きな負担にならないこと。

目標**3** アプリケーション付随の回帰テストでセキュリティ保証を行えること。

さらに、Railsのアプリケーション開発の中で使用するツールとして、次の４つの要求が考えられる。

要求**1** 軽量である(ツールの実行時間が短い)こと 。

要求**2** ツールの設定等がソースコードレポジトリで管理できること。

要求**3** セキュリティ知識がなくても、ツールの指示により適切な修正が行えること。

要求**4** ツールの導入や保守が容易なこと。

要求1については、コマンド抽象化ライブラリを用いたソースコードからのモデル生成の高速化で対応している。要求2については、コマンド抽象化ライブラリや、セキュリティ要求、テスト計画の記述全てに、テキスト形式を用いることで対応した。初期のツール実装ではJSON[1]を用いたが、より記述の負担が少ないYAML[2]も現在は利用可能である。テキスト形式を用いることで、ソースコードのレポジトリを用いた差分管理が容易となる。要求3については、問題があった際には、コマンドラインでツールを実行した際に表示されるメッセージに、その修正方法(Remediation)を提示することで対応した。また、ツールを初めて利用する際には、初期の設定ファイルの自動生成を行う。その後、開発者はそれぞれの開発対象に適した設定に段階的に設定を改善する。要求4については、開発者が簡単にツールを導入できるように、Rubyで一般的なパッケージ管理GEM[3]に対応した。アプリケーションの開発者自身によってもツールの修正や変更を実施できるように、ソースコードはすべてRubyで記述した。

次に、RailroadMapが提供する主要な機能を下記に示す。RailroadMapの実行コマンドは一つ、“railroadmap”、 のみであり、開発者はツールの実行オプションを指定することで、これらの機能の実行を選択できる。

[1]JSON, JavaScript Object Notation. http://www.json.org/
[2]http://yaml.org/
[3]https://rubygems.org/

機能 **1** ツールの初期化。

機能 **2** Web アプリケーションのソースコードからのセキュリティ検証モデルの生成。

機能 **3** セキュリティ要求と、実装との整合性のチェック。

機能 **4** 動的テストケースの生成、テストの実行管理、テストカバレッジ計測

機能 **5** セキュリティ知識（CWE,CAPEC）とのコード実装とのマッピング

機能 **6** 外部ツールとの連携

機能 **7** ダッシュボードによるセキュリティ保証の一元管理。

次節で、それぞれの機能について詳細を説明する。

4.1.2 ツールの初期化 (機能 1)

開発者は RailroadMap を最初に実行する際に、初期化作業を行う。まず、対象となる Web アプリケーションの利用パッケージの解析と、MVC コードの特定を行う。Rails の場合、MVC コードはアプリケーション本体以外にも、外部パッケージが提供する MVC コードを用いる場合がある。そのため、利用しているパッケージ[4] をすべて調査し、セキュリティ検証モデルを構築する際に必要なすべての MVC コードを特定する。また、コマンド抽象化ライブラリは、基本的に Rails が用いるパッケージ単位で作成するため、コマンド抽象化ライブラリの選択も、アプリケーションが利用しているパッケージを元に行う。

次に、必要となる設定ファイルの初期化を行う。この際に、Rails が作成するルーティングテーブル情報を抽出する。これは Web アプリケーションの URL と、それに対応する Controler（セキュリティ検証モデルの Controler state に相当）を定義するものである。セキュリティ要求や、モデル生成の補正指示などのテンプレートもこの時に生成される。

4.1.3 セキュリティ検証モデルの構築 (機能 2)

MVC タイプの Web アプリケーションの振る舞いは、図 3.2 で示したように、Controller と View からなる制御フローモデル（状態遷移モデル）で表す。通常の動作では、View で生成されたページがブラウザで表示され、利用者の操作によるリクエストがメッセージとしてサーバーに送られる。送られたメッセージは、フレームワークにより、その URL に対応する Controller に送られる。入力は Controller

[4]開発者は Gemfile に列挙して指定するが、依存関係のあるパッケージも自動的に読み込まれる

で処理された後、対応する View に処理が移る (render)。そのレスポンスとして、Controller に対応する View により、出力メッセージが生成される。リクエストによっては Controller は 別の Controller に処理を移す場合がある (redirect)。これには、攻撃などの不正なメッセージを受信した場合のエラー処理も含む。

本ツールで実現する認証認可などのアクセス制御に関する静的なセキュリティテストには 2 つの視点がある。

ひとつは、外部からのあらゆる不正なアクセスに対処するためのセキュリティ機能 (PEP、Policy Enforcemnt Point) が配備されていることの確認と、ポリシー定義の実装 (PDP、Policy Decision Point) が正しいことの確認である。これについては、Web アプリケーションのアタックサーフェースである Controller への入力、攻撃も含む全ての入力について、アプリケーションが正しく振る舞うことを確認する。この問題に対しては、静的検証を用いて、PEP の配置をセキュリティ要求と照らし合わせてチェックする。

もうひとつは、セキュリティ機能の配備に付随して発生する、セキュリティ的には問題ないが、アプリケーション的にはバグとなる問題の検出である。利用者が正しくアプリケーションを利用しているのに、アプリケーションがセキュリティ上の警告を発してしまうことがあげられる。具体的には、管理者権限へのリンクを誤って表示してしまい、一般ユーザーがそのリンクをクリックした際に、権限違反の警告を発してしまう事などである。こうした問題は Navigation error [37] と呼ばれる。この問題については、制御フォローモデル上で、状態遷移前後で権限が一致していることを確認することで、検知することができる。Rails のコード生成機能を用いると、すべての遷移パターンが生成される。そのため、セキュリティ要求（アクセス制御）がある場合には、開発は自動生成されたコードの修正を行わなければならない。この修正が不十分だと、Navigation error が発生する。

RailroadMap のこれらの静的検証の機能は、次の 3 つのステップで実装されている。

Step 1: ソースコードからセキュリティ検証モデルの生成

Step 2: セキュリティ検証モデルへのポリシーの注入

Step 3: セキュリティ検証モデルを使った静的検証（機能 3）

RailroadMap 内部でのこの処理の流れを、図 4.1 に示す。ステップ 1 で、コマンド抽象化ライブラリを用いてソースコードからセキュリティ検証モデルを生成する。ここではセキュリティの確認に必要十分な情報（コード由来のセキュリティポリシーを含む）がモデルに反映される必要がある。この際に、コマンド抽象化ライブラリの不備や、対象コードで使用している Ruby 記述がサポート外であった場合には、警告 (Warnings A) が報告される。ステップ 2 では、生成されたモデルに対して、セキュリティ要求（開発者定義の要求に由来するセキュリティポリシー）を

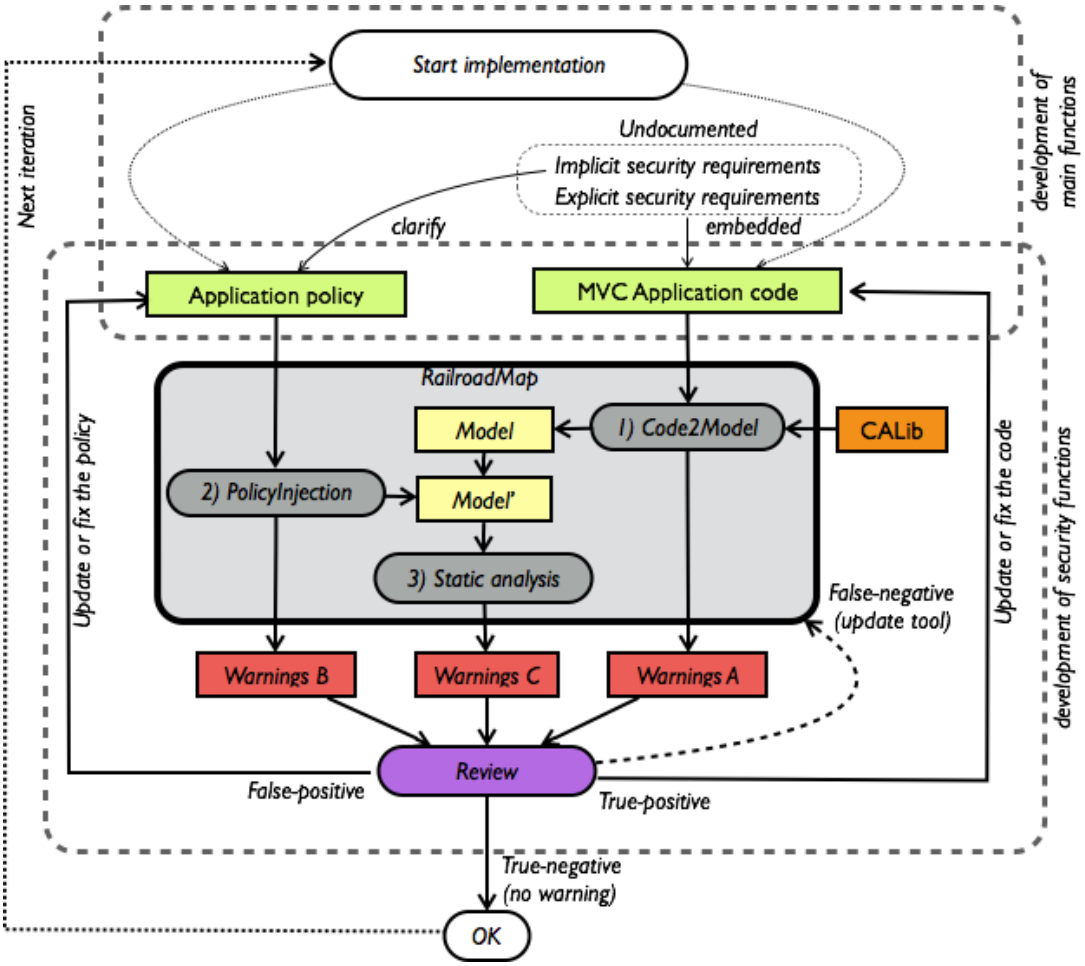

図 4.1: RailroadMap の静的検証フロー

展開する。この際に、ポリシー記述と実装に不一致があり、ポリシー展開がうまくいかない場合には、警告 (Warnings B) が報告される。ステップ 3 では、コード由来と要求由来のポリシーを比較し、異なる場合に警告 (Warnings C) を発する。また、制御フローモデル上で状態間のポリシーの整合性を確認し、問題 (Navigation error) がある場合は警告 (Warnings C) を発する。

このように、コードの実装と、セキュリティ要求の定義を独立して扱うことで、コードの変更にともなうポリシーの修正、もしくは要求の変更にともなう実装の修正など、様々な変化に対応できる。初期段階では、多くの警告が発生するが、それを修正することで、要求やコマンド抽象化ライブラリが成熟し、対象アプリケーションのセキュリティ保証が完成する。次にステップ 1 と 2 について詳細を解説する。ステップ 3 (機能 3) については 4.1.4 節で詳しく説明する。

4.1.3.1 ソースコードからセキュリティ検証モデルの生成 (Step 1)

Rails の場合、セキュリティ検証モデルの状態とアプリケーションコードとの対応は、ファイルやメソッドが状態と一意に対応するために、特定が容易である。実際のアプリケーションの MVC を構成するコードは、様々なコマンド (API) を呼び出し、様々な処理を行っている。これらのコマンドの実態は、プログラムのクラス内で定義された関数、Helper 関数、フレームワークで定義される関数である。

Rails はフレームワークも Ruby で記述されており、フレームワークすべてのコードをパースしてモデルを構築することも不可能ではないが、対象とするコードサイズが膨大となり、2.5.8 節で指摘したように軽量なツールにはならない。ここで、確認したいのはセキュリティに関するアプリケーションの振る舞いであり、これらは基本的に MVC コード上に表現されている。そのため、すべてのコードを検査対象とする必要はない。そこで、MVC の状態の抽出とその相互作用（振る舞いとデータフロー）を検出し、セキュリティ機能が正しく配置されいるかを確認できる最低限のソースコードを対象とすることが、提案手法が取る現実的なアプローチである。

このアプローチを実現するのが、コマンド抽象化ライブラリである。ここで、Rails の MVC コード部分が利用する各種のコマンドの性質を抽象化する。例として、表 4.2 に Rails で用いられる一般的なコマンドと、その分類を示す。分類 1 では、コマンドがモデル生成に関与するか、セキュリティに関するコマンドかを分類する。モデル生成の場合、分類 2 で制御フローとデータフローに関するコマンドかを決める。セキュリティの場合は、あるセキュリティ機能を実現する Security Command (SC) と、その使用が脆弱性の原因となる Risky Command (RC) に分類する。これらの分類は排他的ではなく、独立して定義できるコマンドの属性である。

例えば、link_to コマンドは、表示されるページにリンクの URL を表示する。つまり、View から Controller への遷移となる。コードにおけるこのコマンドの実行条件が制御フローモデル上での遷移条件 (Guard) となる。RailroadMap のツールとしてサポートするコマンド抽象化ライブラリには、Rails やセキュリティ機能に関する主要なパッケージが提供する基本的なコマンドが含まれる。アプリケーション開発側が独自で作成したコマンドについては、アプリケーション開発側でコマンド抽象化ライブラリを作成する。

本ツールが対象とするソースコード領域とモデル化の流れは、3 章の図 3.3 のようになる。まず、MVC を構成するアプリケーションコード (./app/* 以下に配置されている) をすべてパースし、一旦 AST に変換する。ツールはこの AST を用いてコードを解析し、セキュリティ検証モデルの生成を行う。AST 上に現れたコマンドは、コマンド抽象化ライブラリの定義参照して、制御フロー、データフローを構築する。

3 章の図 3.2 で示すように、制御フローの各状態 (State) は、MVC コードの View

表 4.2: Rails の代表的なコマンドとその分類

分類 1	分類 2		コマンド例
モデル化	制御フロー		redirect, render, link_to, submit
	データフロー		label, *_field
セキュリティ	Security Command (SC)	認証	authenticate_user!, require_user
		認可	authorize!, role_required, has_role?
		エスケープ	h
	Risky Command (RC)		raw, html_safe, eval

表 4.3: Rails のアプリケーションソースコードと制御フローモデルとの対応

Code type	File	Scope	Control flow (Navigation) model
Model (CanCan)	app/models/ability.rb	class (par file)	$r \in R, a \in A, o \in O$, (Role,Asset,Operation) $ACL[r][a][o]table$
Model (Ruby/ORM)	app/models/[a].rb	class (par file)/code	$v \in V$ (Variables)
Controller (Ruby)	app/controllers/[a]_controller.rb	method (= o)	$s \in S_C$ (Controller state)
		command (redirect) command (render) command (sec. funds) variable	$t \in T_{CC}$ (Transition from C to C) $t \in T_{CV}$ (Transition from C to V) g (Guard of trans. and attribute of the state) $v \in V$ (Variables)
View (template, ERB)	app/views/[a]/[o].erb	file	$s \in S_V$ (View state)
		condition and command (link_to, submit)	$t \in T_{VC}$ (Transition from V to C) with g (Guard of trans.)

(アプリケーションコードの ./app/views/*/*.erb もしくは *.haml ファイル) と、Controller の場合は Class ファイル (./app/controllers/*_controller.rb) の各メソッドが対応する。制御フローの状態遷移 (Transition) は、表 4.2 で示した View や Controller コード内のコマンドにより生成される。この場合、遷移元は遷移コマンドを呼んだ状態である。遷移先はコマンドの引数などから求める。遷移先の指定は、アプリケーション内ではパス名が指定される。初期で作成したアプリケーションのルーティングマップでこのパス名と状態との対応は特定できるため、モデル上での遷移先はコードの静的解析で容易に特定できる。実際の内部処理では、すべてのコードをパースしなければ状態が確定しないため、一旦すべての MVC コードをパースして状態と遷移元を確定した後に、遷移の遷移先を確定する。アプリケーションのソースコードと制御フローモデルの対応関係を表 4.3 に整理する。

データフローモデルもコマンドをベースに、ソースコードから抽出される。(Controller や View の) 状態と Model (もしくはその逆) とのデータの流れを記録する。また、アプリケーション内部で使用される変数をトレースし、データフローを作成する。ただし、Rails の MVC コードは分散して記述されるため、変数のスコープの特定が難しい。基本的には変数の名前で対応をとるが、精度は完璧ではない。

セキュリティ機能に関するコマンドの場合は、各状態の持つコード由来のセキュリティポリシーとして記録する。

ここでのセキュリティ検証モデルの生成能力は、主にコマンドライブラリ対応状況に依存する。フレームワークである Rails やアプリケーションが利用している各種のパッケージは、かなり頻繁にその機能が更新されており、このライブラリの保守も本ツールの課題の一つである。そのため、MVC コードに現れるコマンドはパース中に記録され、未対応のコマンドを容易に発見、追加できるようにした。

ソースコードから自動的にセキュリティ検証モデルが生成できることが理想であるが、実際には遷移先が本パーサーで容易に特定できない場合が生じる。その際には警告を発するとともに、設定ファイルを準備することで、手動で遷移先を補足できるようにした。

4.1.3.2 セキュリティ要求の記述とセキュリティ要求モデルへの注入 **(Step 2)**

本ツールは、代表的な認証認可のモジュールの提供するコマンドと、その振る舞いをライブラリとしてサポートすることで、セキュリティ検証モデルにそれらの振る舞いを反映させている。

Rails でのアクセス制御の実装には、2.5.5 節で示したように、アプリケーションが独自に実装する場合と既存のパッケージの利用とがある。認可の実装方法、つまりポリシーの記述に関しては、CanCan のようにポリシーをコード中に埋め込む方法と、データベースを使い柔軟に設定できるようにする方法とがある。CanCan の場合、当初はそのポリシー記述からポリシーを抽出する方法をとったが、実際のアプリケーションでは不明確な定義が多く見受けられた。これは、ポリシーを Ruby の条件文で自由に記述できることが原因である。それに対し、TheRole は、データベースのテーブルで定義するため曖昧さは少なくなるが、データベースにポリシーを設定するまでは分析ができない。独自実装の場合は、ソースコードやデータベースの内容による分析を汎用的にサポートすることは困難である。そこで、本ツールでは、アクセス制御ポリシーも含め、最低限のセキュリティ要求を独自に定義する。また、CanCan や TheRole のような既存パッケージの場合は、本ツールで利用パッケージ向けのポリシー定義（コードやデータベースの初期値）を自動生成することも可能である。

ポリシーの記述は、初期の実装では Ruby（によるハッシュ定義）を用いたが、最新の実装ではツールへの入力は JSON に統一した。実例を 4.2 節の Listing 4.3 に示すが、個々のモデルに対して、認証認可の有無、RBAC の場合はそのアクセス制御リストの定義、HTML や状態遷移図で出力する際の色などを指定する。

一般的に、この定義が複雑になるのは認証認可に関する部分が主であり、既存の認証認可パッケージを利用する場合は、対応するテンプレートを利用することで開発者の負担は低減されると考える。

生成されるセキュリティ検証モデルのクラス図を図 4.2 に示す。制御フローモデルはアプリケーションの全体の振る舞いを示す状態遷移図となる。データフローモデルはアプリケーションへの入出力を中心としたデータの流れの記録である。実

際には、双方のモデルはクラス図にあるようにツール内部でインスタンスを共有している。

セキュリティ検証モデル上では由来の異なる 2 つのポリシーを管理している。ひとつは、ソースコードから抽出されたもの、もう一つは、要求定義から生成し、各状態に付与されたものである。これらは、状態 (State)、変数 (Variable)、モデル (Model) に割り当てられる。この 2 つのポリシーの整合性をチェックすることで、セキュリティ要求と実装の各種整合性を確認するのが、本ツールの大きな特徴である。

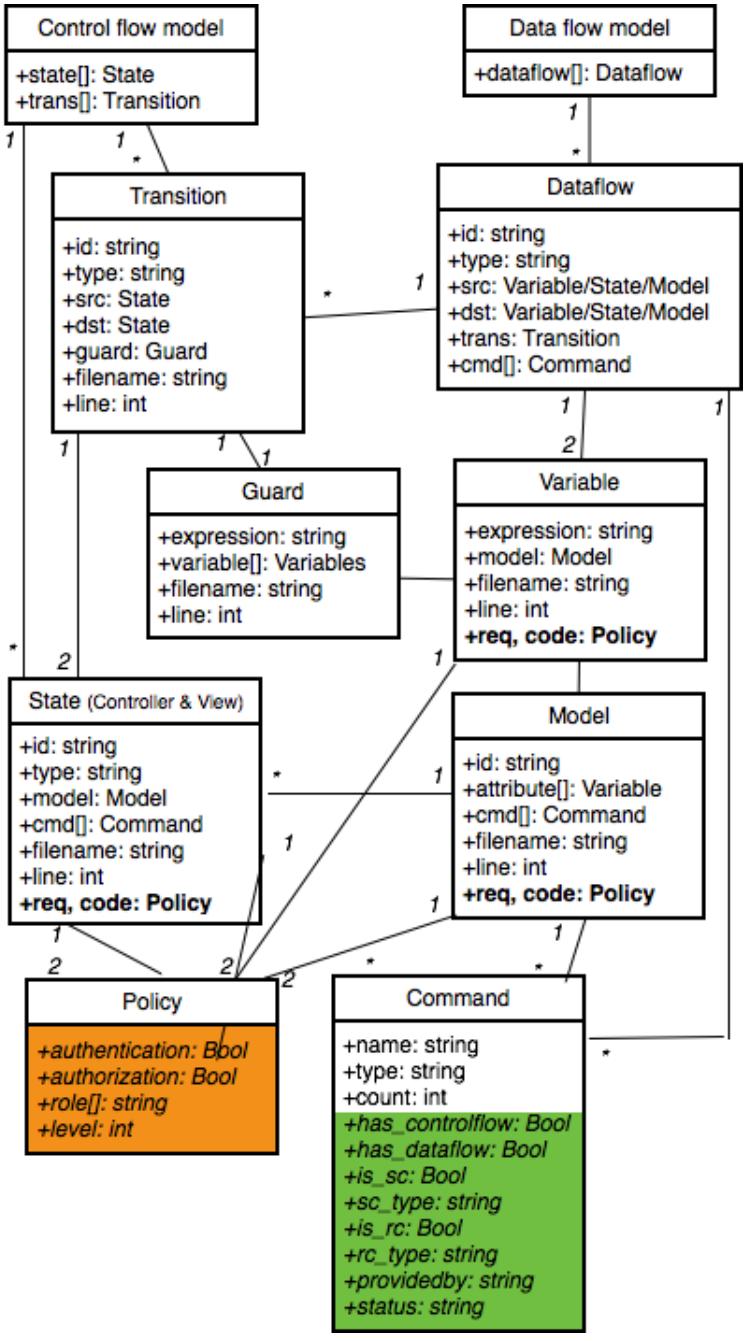

図 4.2: 生成されるセキュリティ検証モデルのクラス図

4.1.4 静的セキュリティテスト (機能 3)

次に、生成されたセキュリティ検証モデルを用いて、アクセス制御機能の実装と、セキュリティ要件との一貫性をチェックする。このチェックで報告される警告の種類、深刻度、問題の発生箇所を表 4.4 にまとめる。CPD は CanCan のポリシー記述コードが複雑で、ポリシーが正しく抽出できていない場合に報告される。IP1,IP2 はコード上にセキュリティ機能配置されているにもかかわらず、ポリシー定義が無い場合、MT1 は要求にも実装にも明確なポリシーが無い場合、MT2 は要求はあるが、実装にセキュリティコマンドが配置されていない場合、MT3 はポリシー記述はあるが、要求、実装に無い場合、MZ1 はポリシー記述はあるが、実装にコマンドが配置されて無い場合、MZ2 はポリシー記述と View でのチェックはあるが、Controoller に認可のコマンドが配置されて無い場合、IN は View で認可がチェックできていない場合（ナビゲーションエラー）である。次に、ツールが実行する静的検証の詳細を示す。

表 4.4: 警告の種類と深刻度

Warning	Sever-rity	Req.	PDP	PEP	Nav.
Complex policy definition (CPD)	low	–	x	–	–
Improper policy 1 (IP1)	high	–	x	o	–
Improper policy 2 (IP2)	high	–	x	–	o
Missing authentication 1 (MT1)	low	x	x	x	–
Missing authentication 2 (MT2)	high	o	-	x	–
Missing authentication 3 (MT3)	mid	x	o	x	–
Missing authorization 1 (MZ1)	high	–	o	x	–
Missing authorization 2 (MZ2)	high	–	o	x	o
Improper navigation (IN)	mid	–	o	o	x

4.1.4.1 セキュリティ検証モデルを用いた静的検証

まず、PEP の配置についてのチェックは、この 2 つのポリシーを比較することで容易に実現できる。不一致があった場合は、定義されたセキュリティ要求、もしくはコードの実装が間違っている可能性があるが、どちらが原因かはツールにはわからない。したがって、ツールが発する警告を開発者がレビューし、意図するセキュリティ要求と実装を決定し、修正を施す。この過程で、アプリケーション

の持つセキュリティ要求と実装が明確に定義されることになる。例えば、こうした作業が発生の例として、認証認可が不要な Controller の確定が挙げられる。認証認可が不要な場合は、Controller には該当するコマンドは記述されないが、それが意図したものか、記述ミスなのかはコードからは読み取れない。そこで、セキュリティ要求の形で明示することで、実装コードの背景となる暗黙の仕様を文書化する事ができる。

Algorithm 1 PEP check.

Require: ACL, S ▷ Nav. model
Require: Q ▷ Requirements
Require: $W \leftarrow 0$ ▷ Warning

procedure CheckPEP
  for each $s \in S_C$ **do**
    $res1 \leftarrow hasAuthenticationCheck(s)$
    $res2 \leftarrow hasAuthorizationCheck(s)$
    $res3 \leftarrow hasPolicyDefinition(s)$
    if $\neg res1 \& \neg res2 \& \neg \exists s \in Q$ **then**
      $W \leftarrow W \cup NewWarning(MT1)$
    else if $\neg res1 \& \exists s \in Q$ **then**
      $W \leftarrow W \cup NewWarning(MT2)$
    else if $\neg res1 \& res3 \& \neg \exists s \in Q$ **then**
      $W \leftarrow W \cup NewWarning(MT3)$
    end if
    if $res3 \& \neg res2$ **then**
      $W \leftarrow W \cup NewWarning(MZ1)$
    end if
  end for
end procedure

次に Navigation error の検出方法について述べる。これは遷移の前後でセキュリティ要求が異なる場合に発生する。具体的には、複数のロールによって共有されている View でそのユーザーのロールによって異なる遷移先を指定する必要がある場合に、その確認とロールに対応したページの生成を忘れている場合である。Rails には MVC コードの自動生成機能 (Scaffold) があり、広く利用されているが、セキュリティ要求への配慮がないコードが自動生成されるために Navigation error は発生しやすい。アプリケーション的には脆弱性はないが、通常の使用で権限エラーが出てしまうのはアプリケーションのバグである。セキュリティ検証モデルは すべての View と Controller 間の遷移と遷移条件、そのセキュリティ要求を記録しており、前後のセキュリティ要求が異なる場合は、ロールの確認等の適切な遷移条件が与えられているかチェックを行い、問題がある場合は警告を出す。

最後に、セキュリティ要求の定義が、セキュリティ検証モデル上で矛盾が無いかを検証する。これにはデータフローモデル l を用いる。

一般的には、MVC の構成する各状態は、共通のセキュリティ要求のもとにグループを構成している。ただし、異なるセキュリティ要求との間でデータの流れが存在する場合、それが意図したものなのか、バグなのかを判別したい。ロール定義だけでは確認が難しい。

図 4.3 に一例を示す。この場合、C1,V1,M1,S1 はセキュリティ要求の高い管理者

Algorithm 2 Navigation check.

Require: R, ACL, S, T ▷ Nav. model
Require: $ss \in S_V$ ▷ Start state
Require: $r \in R$ ▷ Role
Require: W ▷ Warning
procedure CheckNAV(ss, r)
  for each t **in** $GetTransFrom(T_{VC}, ss)$ **do**
    $sd \leftarrow GetDstState(t)$
    if $isVisited(sd, r)$ **then** ▷ detect infinite loop
      ↵ ▷ escape
    else
      $setVisited(sd, r)$ ▷ set flag
    end if
    $g \leftarrow GetGuard(t)$
    $o \leftarrow GetOperation(sd)$
    $a \leftarrow GetAsset(sd)$
    $res1 \leftarrow EvaluateGuard(g, r, o, a)$ ▷ Trans
    $res2 \leftarrow EvaluatePEP(sd, r)$ ▷ State
    if $res1 = deny$ **then** ▷ guard: AC1
      if $res2 = deny$ **then** ▷ end of transition
        ↵
      else ▷ inconsistent V and C
        $W \leftarrow W \cup NewWarning(IN)$
      end if
    else if $res1 = allow$ **then** ▷ guard: AC1
      if $res2 \neq allow$ **then**
        $W \leftarrow W \cup NewWarning(MZ2)$
      end if
    else if $res1 = true$ **then** ▷ guard: AC2
      if $res2 \neq allow$ **then** ▷ improper guard
        $W \leftarrow W \cup NewWarning(IN)$
      end if
    else if $res1 = false$ **then** ▷ guard: AC2
      if $res2 \neq deny$ **then** ▷ improper guard
        $W \leftarrow W \cup NewWarning(IN)$
      end if
    else ▷ no guard
      if $res2 = allow$ **then**
        $W \leftarrow W \cup NewWarning(IN)$ ▷ FP?
      else if $res2 = deny$ **then**
        $W \leftarrow W \cup NewWarning(MZ2)$
      end if
    end if
    ↵CheckNAV(sd, r) ▷ recursive call
  end for
end procedure

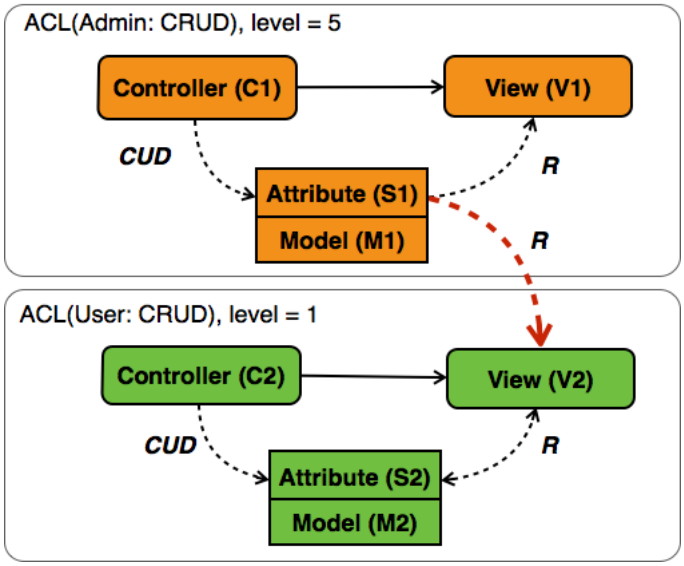

図 4.3: Dataflow model を使ったポリシー整合性のチェック

が管理しているデータに関する状態である。一方、C2,V2,M2,S2 は一般ユーザーのデータに関する状態である。実際のデータフローは Model の Atttibute レベルで起こる。ロール定義だけではデータのセキュリティレベルはわからないため、セキュリティ要求で指定したレベルを用いて、データフローを解析する。この例では、S1(level=5) から V2(level=1) へのデータフローは、 高いレベルから低いレベルへのデータフローのため警告となる。意図するものであるなら、Atttibute S1 のレベルを 1 に、セキュリティ要求のなかで定義することで意図した実装であることを明示できる。意図しない場合は、該当するコードを削除する。

4.1.5 動的セキュリティテスト (機能 4)

2.2.3.3 節で述べたように、Web アプリケーションの安全性を確認する最終的な手段はペネトレーションテストである。人手によるペネトレーションテストや、Web アプリケーション向けの脆弱性スキャナーを用いたペネトレーションテストは、実施に時間が掛かること、カバレッジが不明なこと、問題の修正に手間が掛かることが課題である。

Rails の開発では、テスト駆動開発 (TDD, BDD) の環境が充実しており、ほとんどの Web アプリケーションへの攻撃を、実行可能なユーザーアクセプタンステスト（UAT）で記述することが可能である。ただし、網羅的なセキュリティテストケースを人手で作成し、保守することは現実的でない。そこで、テスト駆動開発と連携して効率的なセキュリティテストを実施するために、以下の 3 点が重要であると考える。

- セキュリティテストケースのカバレッジ計測
- 必要最小限のセキュリティテストケースの選択
- セキュリティテストケースの自動生成
- テスト結果の統合

4.1.5.1 セキュリティテストケースのカバレッジ計測

TDD をセキュリティに対する回帰テストで活用することができれば、プログラマが主体的にセキュリティの確保に参加できる。従来の動的なセキュリティテスト手法は、Web アプリケーションスキャナーのように、アプリケーションに対して網羅的なブラックボックステストを実施する。アプリケーション規模が大きくなると、セキュリティテストの適用箇所が増加し、テストの組み合わせが増加する。結果として実行時間が掛かる。したがって、プロググラマが日常のコーディングの中で用いるセキュリティツールではない。TDD ではアプリケーションの内部状態を参照することでより効率的に、つまり少ないテスト数でセキュリティテストを実施できる可能性がある。これは一般にホワイトボックステストと呼ばれる。課題は、必要最小限の数のテストケースを配置する手法である。本研究では、SINK カバレッジを用いることでこの問題に対処する。SINK の定義はコマンド抽象化ライブラリで行う。

TDD と本ツールの関係を図 4.4 に示す。エンジ色の部分がツールの実行とその入出力を示す。アプリケーションコードはコマンド抽象化ライブラリを用いてセキュリティ検証モデルに変換される。その際に、SINK の一覧も作成する (Parser 1)。次に、TDD 側のテスト実行の結果、得られるコードカバレッジと、コード中の SINK を比較し、それぞれの SINK のテストの有無を確認する (Sink analysis)。

最後に、TDD 側のテストコードをパースし、テストケースで用いられるアクセス制御ポリシーの抽出と、テストケースと SINK との対応を求める (Parser 2)。Sink test coverage はセキュリティコマンドベースのテストカバレッジであり、SC と RC で別々に集計される。Asset test coverage はアクセス制御テストカバレッジである。

メトリックスは全体の把握には適しているが、個別の問題の修正には情報として不十分である。そこで、トレース機能を用いて、特定の警告に関して、ツール実行時の詳細情報を提示する機能を追加した。

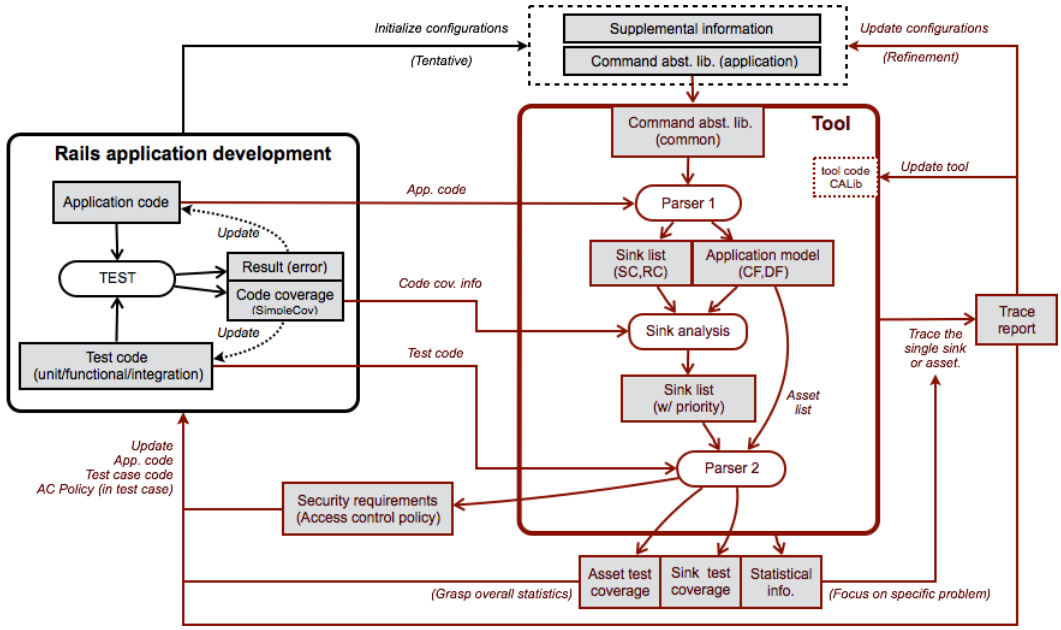

図 4.4: ツールの実行フロー とテストカバレッジの計測

4.1.5.2 セキュリティテストケースの自動生成

セキュリティテストケースを自動生成するためには、対象となる Web アプリケーションのアタックサーフェースでの振る舞いがわかっている必要がある。RailroadMap が生成する制御フローモデルは Web アプリケーションの振る舞いをモデル化しており、UAT の自動生成に適している。

次に、セキュリティテストケースの選択であるが、アプリケーションが利用しているセキュリティ機能 (コマンド) が同じであれば、テストする箇所は最低一箇所で十分である。したがって、個別のセキュリティ機能について、その必要十分なテストパターンを定義し、アプリケーション上の適切な場所をサンプリングしてテストケースを生成すれば、最低限のテストケースで済む。コマンド配置の網羅性については静的に検証できるので、セキュリティ機能の実際の動作が、期待したものであることを動的なテストで確認する。

また、静的検証で問題、もしくは疑わしさが指摘された場合には、その確認のために動的テストを実施する。例えば、フレームワークの提供するセキュリティ機能をバイパスし、独自の実装を行った場合などである。

図 4.5 に Rails Web アプリケーションのセキュリティ検証モデルを用いた動的セキュリティテストフローを示す。Step 1 から Step 2 までは前述の静的検証のフローである。

セキュリティ検証モデルには、テストケース生成に必要なサンプルデータが含まれないため、“Test plan” として、ツールにテストケース生成で必要となる情報を補足する。テスト結果が陽性だった場合は、コードを修正して指摘された脆弱性を排除する。このセキュリティテストケースは UAT の一つとして提供されるため、脆弱性の修正は容易である。陽性であっても、実用上は問題ないと判断した場合は、アプリケーションに動作の前提条件（環境）を整理し、セキュリティ要件を更新する。

初期のバーションでは、テストケース生成に、形式手法の反例を用いる方法を検討した。具体的には、制御フローモデルを B Method の状態遷移図で出力し、入出力条件から、反例として、特定の遷移経路を得て、それを UAT に変換する。しかしながら、実際には制御フローモデルを直接検索し遷移経路を抽出する手法で実用上は十分なこと、また UAT の記述スタイルに自由度が多く、個別のアプリケーションに適したテストケースの生成が難しいことから、最新のバージョンでは形式手法の反例は利用していない。代わりに、Testplan の形で、個別のアプリケーション開発のテストスタイルを取り込むようにしている。

Testplan の実例については 4.3.2 節で示す。現在、Testplan によるテスト生成には表 4.5 で示す 3 種類の設定方法をサポートしている。

表 4.5: Testplan の設定

生成方法	説明
automatic	自動生成、開発者は自動生成に必要なパラメータを設定
Modification of existing testcase	攻撃箇所に対する通常の成功シナリオを与える、攻撃シナリオは与えられたシナリオを元に自動生成。
Use existing testcase	Testplan でシナリオを記述

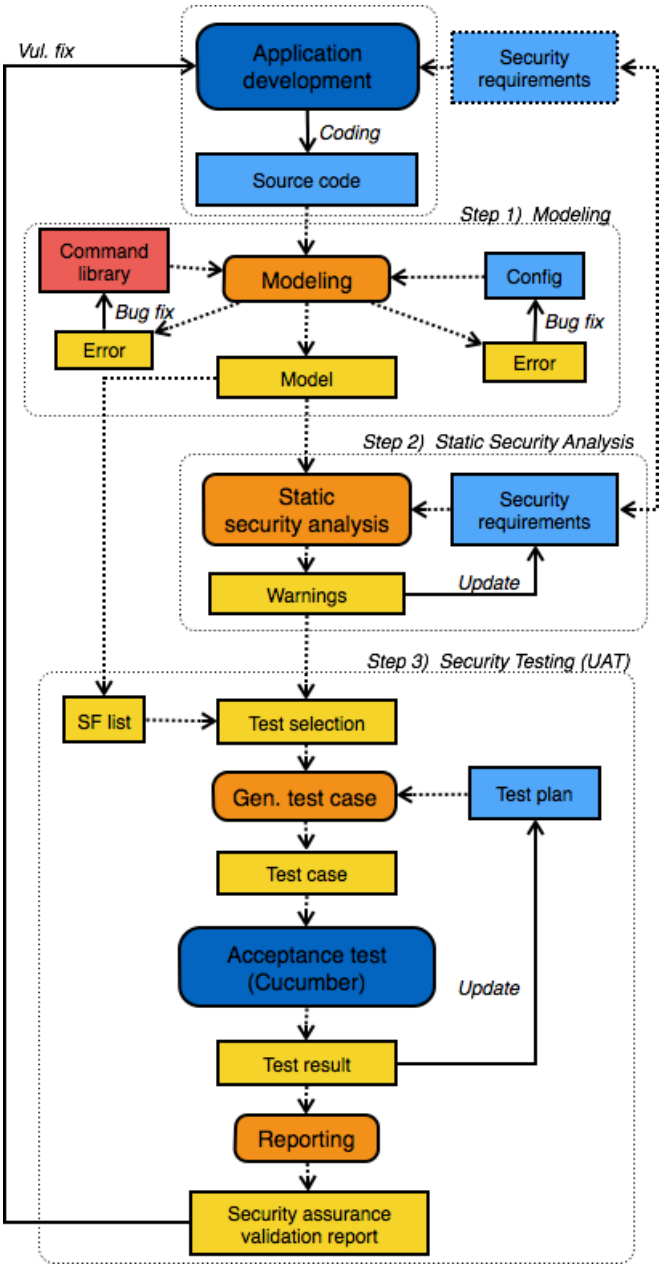

図 4.5: Rails Web アプリケーションのセキュリティ検証モデルを用いた動的セキュリティテストフロー (version 0.2.3)

4.1.6 CWE,CAPEC と実装とのマッピング (機能 5)

CWE,CAPEC とコマンド抽象化ライブラリとの統合の方法については 3.4 節で示す通りである。このツール化では、CWE,CAPEC の情報の取り込みと、選択を機能化する。Rails に該当する CWE,CAPEC の情報の選択は、ツール側の開発で実施する。CWE,CAPEC の情報は XML の形でも公開されており、これをダウンロードしてツールの入力とする。これをツール内部で木構造の有向グラフとして構築し、ツリーの選択と除外を行う。一連の作業は、ツールのテストケースに含む形でスクリプトを用いて実行する。具体的な適用は 4.5 節の実験で示す。

4.1.7 外部ツールとの連携 (機能 6)

近年、Rails の開発で利用可能なセキュリティテストツールが増えてきている。単一のセキュリティテストツールですべての問題をカバーすることは難しいため、利用可能なツールは積極的に利用してゆくことが望ましい。表 4.6 に RailroadMap の Version 0.2.4 で対応している外部ツールの一覧を示す。

表 4.6: 外部セキュリティテストツール

ツール	タイプ	概要
bundle-audit	静的	パッケージの既存脆弱性チェック
Brakeman	静的	様々な問題の静的検証に対応
Codesake-dawn	静的	様々な問題の静的検証に対応
skipfish	動的	Web アプリケーション用の脆弱性スキャナー
W3AF	動的	Web アプリケーション用の脆弱性スキャナー
ZAF	動的	Web アプリケーション用の脆弱性スキャナー

静的セキュリティテストツールの場合は、その実行結果をダッシュボードに取り込む。動的セキュリティテストツールの場合は、実行に際して認証情報の設定などを与える必要がある。本ツールはセキュリティ要求を把握しているため、そうした設定を自動生成することが可能である。そこで、ツール実行のための設定や、実行スクリプトを生成するとともに、実行結果をダッシュボードに取り込む。

4.1.8 ツールのアウトプット (機能 7)

本ツールの警告は、コマンド実行時に標準出力に表示される。また、HTML でも出力される。初期のバージョンでは B Method での出力にも対応したが、最新版では、生成したモデルを JSON の形でエキスポートすることが可能である。

4.1.8.1 ダッシュボード

HTML は制御フローモデル 、データフローモデルなどの情報をテーブルの形で表示する。DataTables[5] を利用しているので、簡単なソートやサーチも HTML 上で利用できる。図 4.6 に例を示す。警告タブでは、すべての警告が列挙される。エラータブは、モデル生成時の問題点など、ツール自体の問題を列挙し、コマンドタブでは、対象のアプリケーションで利用されているコマンドの詳細とその頻度が列挙される。また、アセットタブでは、Web アプリケーションの Model、Controller、View、Policy の一覧が列挙される。制御フローモデル (旧のバージョンでは Navigation model と表示) タブは MVC の状態遷移のエッジが列挙される。データフローモデルタブではデータフローが列挙され、最後に、UAT タブでは、Test selection、テストケースの生成状況、実行結果が列挙される。

4.1.8.2 状態遷移図

制御フローモデルを Graphviz[6] を使い状態遷移図としてプロットすることもできるが、状態数が多くなるとあまり実用的ではない。そこで、ポリシーの展開状態 (M to C to V) に、View と Controller 間の遷移を追加したポリシー図を PDF で出力している。

また B Method の形で出力することも可能である。B Method のモデルチェッカーである ProB[7] 上での、アプリケーションの動作の確認例を図 4.7 に示す。

[5] http://datatables.net/
[6] http://www.graphviz.org/
[7] http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page

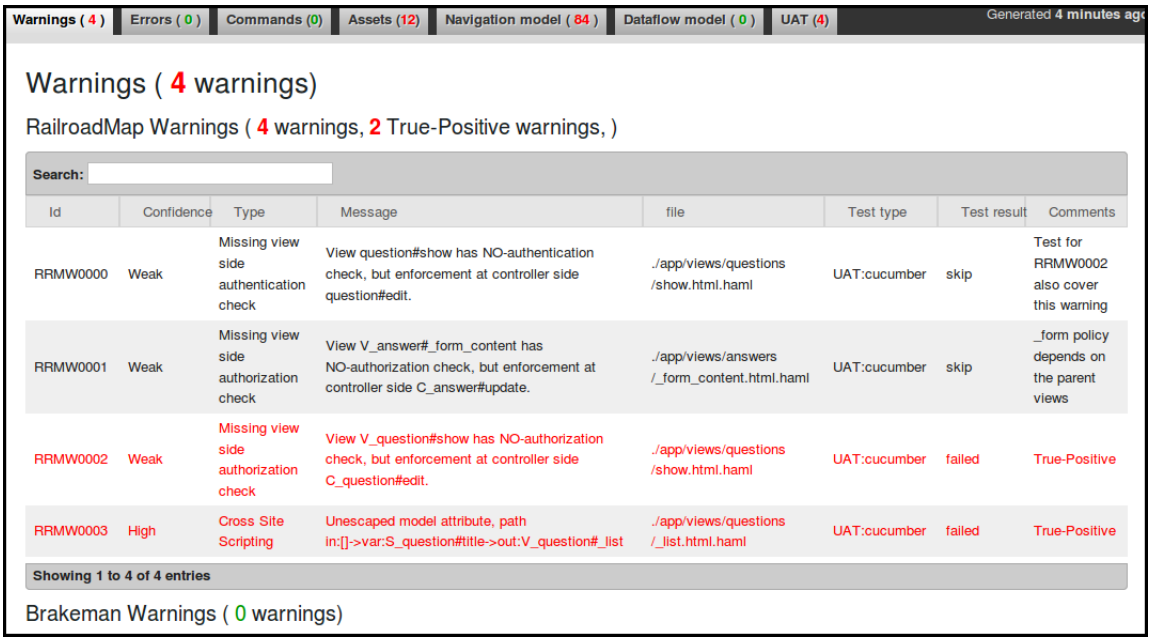

図 4.6: ダッシュボードの例 (version 0.2.3)

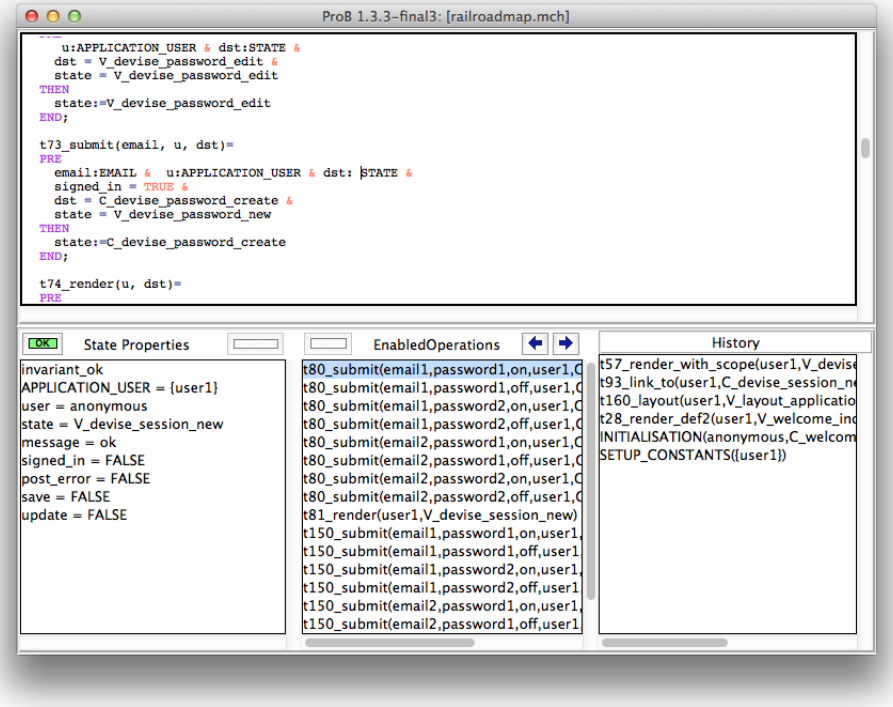

図 4.7: B Method での状態遷移の動作シミュレーション

4.1.9 RailroadMap の更新履歴

表 4.7 に RailroadMap の実装履歴と機能評価との関係をまとめる。RailroadMap はまだ開発途上であり、3 章で掲げた提案を実験的に実装している。こうしたツール開発自体も、アジャイルソフトウェア開発の対象であり、テスト駆動開発で機能の追加を行っている。

本ツールの開発の開始時点では、Rails 向けの静的検証ツールが無く、静的検証ツールとしての機能実装を進めたが、その後、Brakeman などの静的検証ツールがリリースされたため、バージョン 0.2.0 から従来の静的検証ツールでは対応が難しいセキュリティ要求を用いた実装検証ツールとしての機能を追加した。バージョン 0.2.2 からは、それまで実装コード上に分散していたセキュリティ情報をコマンドライブラリの形に集約し、実装コード側での処理をより汎用的になるように、実装方式の変更を行っている。それ以降は、3 章で掲げた提案を実験的に実装し機能評価を進めてきた。

本ツール自体 Ruby で記述しているため、アプリケーション開発者が自由に機能の修正や拡張を行うことが可能である。より実用的なツールにするためには、コマンド抽象化ライブラリの効率的な保守、包括的なセキュリティ保証ケースの整備、ユーザーインターフェースの改善が必要である。ツールの課題と改善については 5.3 節にまとめる。

表 4.7: RailroadMap の変更履歴とロードマップ

Version	Date	変更	実験との対応
0.0.0	2011/03/03	初期バージョン	
0.1.0	2012/03/15	AST ベースのパーサーに入れ替え	
0.2.0	2013/11/05	再設計、セキュリティ要求とコードの整合性チェック方式に変更 HAML に対応、コマンドをライブラリ化	機能評価 (1)
0.2.2	2013/12/12	テストケース生成機能更新	機能評価 (2)
0.2.3	2014/01/11	コマンドライブラリ更新、Dashboard の更新等	
0.2.4	2014/04	Railsgoat に対応	機能評価 (4), 手法比較
0.2.5	2015/06	SINK カバレッジに対応	機能評価 (3)
0.2.6	2015/10	CWE,CAPEC の連携機能追加	機能評価 (4)

4.2 機能評価 (1) セキュリティ要求と実装の整合性確認

次に本ツールの有効性について、アジャイルな開発プロセスの視点から検証、評価する。

4.2.1 既存のサンプルコードへの適用によるセキュリティ要求の洗い出し

3 章で示したように、本ツールは実装と要求の整合性の確認ツールである。開発の主はあくまでアプリケーションの実装作業であり、セキュリティ要求の作成はそれをセキュリティの視点で補足するものである。

本ツールは任意のタイミングで適用することが可能であるが、セキュリティ機能以外のアプリケーションの機能が完成した段階での適用が望ましい。これは、間違っていたり、無用なコードがセキュリティ検証モデルに含まれると、そうしたコードに対する無駄な作業が発生するためである。また、本ツールの活用を前提とする場合は、開発初期の段階でセキュリティ実装について神経質に気にする必要はない。

本ツールは、図 4.1 に示すように、MVC コードのモデルへの変換、モデル上でのポリシー注入、静的解析、の 3 段階でセキュリティ実装と要求との検証を進めてゆく。

アプリケーションの機能が概ね完成した段階で本ツールを適用し、セキュリティ検証モデルの生成を行う。この際に、MVC コードのモデルへの変換に問題がある場合は、警告（Warning A）が報告されるので、補正する。

次に、ポリシー注入の際に発生する警告 (Warning B) を修正する形でポリシー定義（セキュリティ要求）を作成する。ここでは、主にポリシー定義が正しくセキュリティ検証モデルに注入できるかがチェックされる。たとえば、新しくコードを追加した場合は、ポリシーが未定義警告が出るため、此の段階でポリシー定義を追加する。

最後に、静的検証の結果報告される警告（Warning C）への対処を行う。警告を確認した結果、ポリシー定義が正しく実装側に間違いがある場合（True-positive）はコードを修正する。実装が意図したものであり、ポリシー定義が間違っている場合（False-positive）はポリシー定義（要求側）を調整する。以上の警告がすべて無くなった段階で、セキュリティ要求の定義と実装の確認は終了する。

図 4.1 で示す処理の中での課題は問題が検出できない場合 (False-negative) である。この原因は、モデル化の不備（制御フローモデルやデータフローモデルの生成の不備やコマンド抽象化ライブラリの不備）であり、False-negative の存在がわかった場合はツールを修正する。未知の脆弱性には対応できないが、新しい脆弱性が見つかった段階でツールを更新する。

次に、実際のアプリケーション開発に本ツールを適用し、その活用方法と効果を検証する。Web アプリケーションでは、アクセス制御周りの扱いが最も複雑である。そこで、２つの認証認可のサンプル・アプリケーションを用いて、本ツールの適用を検証した [8] 。

4.2.1.1 rails3-bootstrap-devise-cancan

認証に Devise、認可に CanCan を用いた、代表的なサンプル・アプリケーションである rails3-bootstrap-devise-cancan[9] に対して、本ツールを適用し、この認証モジュールへの対応を確認する。Devise は自身が認証に関する MVC コードを持つため、アプリケーション本体と、Devise の提供する MCV コードから検証モデルを生成する。CanCan の RBAC の定義に従い、セキュリティ要求にポリシーを追加する。作成したセキュリティ要求の詳細は、Github で公開しているチュートリアルとアプリケーションで参照できるため、ここでは特徴的な設定部分について述べる。

ポリシーの記述 (Listing 4.1 に抜粋、version 0.2.0 の Ruby 形式) は、このアプリケーションが使う２つの Model、“user”, “role” に対して設定 ($asset_base_policies) する。“user” Model に関する多くのコントローラーは Devise 側で提供される。そこで、Model から Controller へのポリシー伝搬を自動で行うために、ハッシュテーブル “model_alias” を用いる。認証に関する多くの Controller は、入力側が認証される前の状態になるため、“user” Model に定義した一般ポリシーとは 個別に、Controller のポリシーを設定する。

```
1  $assets_base_policies = {
2    'user' => {      # for Devise and User
3      model_alias: { # for controllers by Devise
4        'devise:registration' => 'user',
5        'devise:session' => 'user',
6        'devise:password' => 'user',
7        'devise:confirmation' => 'user',
8        'devise:unlock' => 'user',
9        'devise:omniauth_callback' => 'user' },
10     is_authenticated: true,
11     is_authorized:    true,
12     level:            10,  # High
13     roles: [
14       { role: 'admin', action: 'CRUD' },
15       { role: 'user',  action: 'CRU', is_owner: true }
16     ],
17     color: 'orange'
18   },
19 <snip>
20
21 $assets_mask_policies = {
22   # Public controllers by Devise
23   'C_devise:session#new' => {
24     is_authenticated: false,
```

[8] 評価で用いたサンプルコードとチュートリアルは現在 Github で準備中である
[9] https://github.com/RailsApps/rails3-bootstrap-devise-cancan

```
25      level: 0,
26      color: 'green' },
27  <snip>
```

Listing 4.1: railroadmap/requirements.rb

このサンプルアプリケーションのテスト結果であるが、まず PEP の配置ミスは発見されなかった。これは検証モデル内の状態遷移の多くが、認証パッケージとしては 成熟した Devise の提供する認証部分であるためである。ただし、Navigation error がひとつ存在することを検出した。これは、認証していない状態でも、ホーム画面にユーザー情報ページへのリンクが表示され、そのため、認証前の Web サイトの利用者がこのリンクをクリックした場合、認証画面に強制遷移しエラーメッセージが表示されることになるためである。その他に、このホーム画面のユーザー情報表示が、データフローの観点から警告された。実際のデータフローに関する警告は 11 あり、多くが 認証周りで発生した False-Positive の警告であった。

図 4.8 は、Devise と CanCan を用いたアプリケーションの振る舞いを表す Navigation model である。図 4.9 は、ツールで自動生成した状態遷移図である。Javascript のライブラリを用いて生成しているが、わかりやすいとは言いがたく、改善の余地がある。

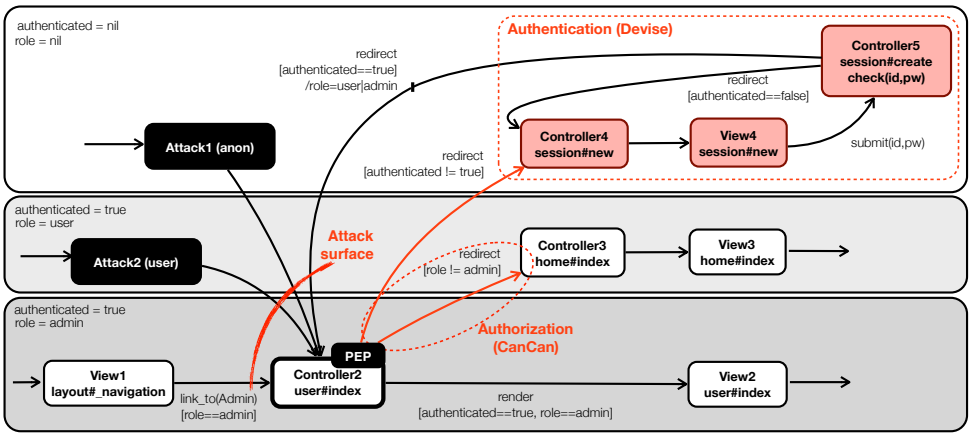

図 4.8: Devise と CanCan の制御フローモデル

アプリケーションのサイズは 4 Models, 26 Controllers, 19 Views, 83 transitions, 113 dataflow となった。この例での実行時間は、 Xeon E3-1265L V2 2.5GH z のマシン上で 0.6 秒以下であり、コマンドラインツールとしての頻繁な利用には支障ないと思われる。

4.2.1.2 the_role example

次に、認可機能を提供する The_ Role に付属しているサンプルアプリケーションに対して本ツールを適用した。このアプリケーションも認証は Devise を利用してい

るため、要求定義は rails3-bootstrap-devise-cancan とほぼ同じである。Devise 同様に The_Role のコントローラーへの Alias を ‘role” Model のポリシー定義に追加する。評価した際のサンプルアプリケーションには、認可に関する Navigation error が 7 箇所発見された。特に Navigation error は ソースコードではわからないため、本ツールの利用は有効である。

アプリケーションのサイズは 3 Models, 48 Controllers, 29 Views, 134 transitions, 170 dataflow となった。この例での実行時間は、Xeon E3-1265L V2 2.5GHz のマシン上で 1.3 秒以下であった。

以上２つのアプリケーションは、CanCan や The_Role といったアクセス制御機能の実装例を示すサンプルアプリケーションであり、例題としての必要最小限のコード量である。しかしながら、実際のアプリケーションでは、アプリケーション開発者はアプリケーション本来の機能を実現する様々な MVC のコードを追加する一連の作業の中で、アクセス制御機能を適切に実装しなければならない。そうした場合に、開発者は本ツールを利用することでアクセス制御に関する実装の不足点やミスを把握できる。結果として、開発者のセキュリティ実装に関係する作業負担を低減し、安全な Web アプリケーションが実装できる。本ツールのユーザーインターフェースにはまだ多くの改善の余地がある。具体的には、アプリケーションのセキュリティ機能のわかりやすい提示方法、問題箇所の提示方法、及び修正の提示方法などである。そうしたユーザーインターフェースを改善することで、より開発者にとって使いやすいツールとなる。

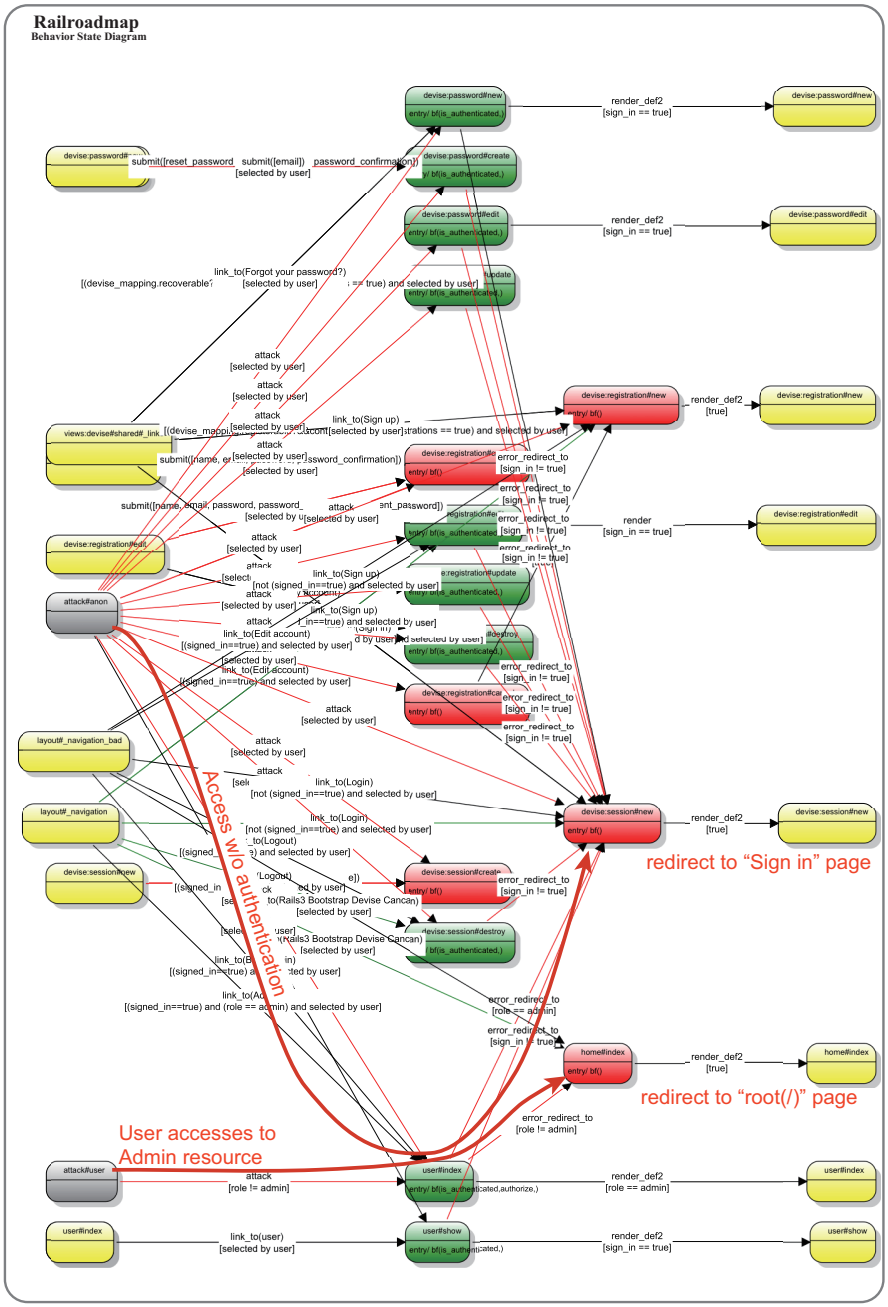

図 4.9: 生成された状態遷移図（抜粋）

4.2.2 既存のアプリケーションの拡張とセキュリティ機能の追加

次に、一度完成したアプリケーションに変更を加える場合を考える。これには２つのシナリオが考えられる。ひとつは、アプリケーションの機能の変更（機能の変更に伴う MVC コードの追加や削除)。もう一つは、アプリケーションのアクセス制御ポリシーの変更（新しくロールを追加するなど、セキュリティ要求の変更）である。前者については、新規のアプリケーション開発と作業の流れは同じである。Rails の開発では、ベースとなる MCV コードは Scaffold と呼ばれる自動コード生成機能で生成される。Scaffold は Model の定義に従ってテンプレートとなる MVC コードを生成するため、開発者は生成されたコードを修正し、アプリケーションを完成させる。アクセス制御が必要なアプリケーションの場合、開発者は自動生成されたコードに認証認可に関するコマンドを配置する。後者については、ポリシー変更に伴い適切な実装コードの変更が必要となる。大規模なポリシー変更の場合は、影響するコードの範囲が大きくなる。Rails の実装では、アクセス制御機能のサポートは MVC コード側で行うため、どちらの場合も開発者が適切に対処しなければならない。

実験では、前節で用いたサンプルアプリケーション、rails3-bootstrap-devise-cancan に対して、表 4.8 に示すように、２つの “Asset (Model)” を追加し、同時に３つのロールも追加することで、アプリケーションの機能の変更とアクセス制御ポリシーの変更が同時に発生するケースについて、ツールが報告する警告がコードの修正作業のなかでどのように変化するか調べる。

表 4.8: アクセス制御テーブル

Role	Assets (Model)		
	All	Article	Comment
Admin	CRUD	-	-
Moderator	-	RD*	RD*
Reporter	-	CRU*	R*
Member	-	R*	CRU*

C: Create, R: Read, U: Update, D: Delete
* 実験追加したルール

次に、Web アプリケーションの開発を段階的に進めながら、セキュリティ機能の適切な実装を、ツールの警告をベースに進めてゆく。表 4.9 に各段階での警告の数を遷移を示す (警告の種類は表 4.4 参照、MT3:認証 PEP 配置ミス、MZ1: 認可の配置ミス、IN：Navigation エラー)。

- Step 0 サンプルアプリケーションにツール適用。オリジナルコードには警告はなく安全な実装である。
- Step 1 表 4.8 に示す “Model” を 2 つ追加する。Scaffold による自動コード生成は認証の PEP を自動配置しない。そのため警告 (MT3:認証 PEP 配置ミス) が 14 件発生する。
- Step 2 CanCan の Ability.rb を更新し、表 4.8 に示すをロールを 3 つ追加する。その結果、認可が必要なモデルが識別されるため、認可のコマンド実装の不在が警告（MZ1: 認可の配置ミス) として 6 件発生する。
- Step 3 以上の警告がなくなるように、Devise 及び CanCan の認証認可に関するコマンドを実装コードに追加（PEP の配置）する。結果として、異なるロール間の遷移（IN：Navigation エラー）が発生し 145 件の警告となる。
- Step 4 ロールに従って適切なページが生成されるように “View” テンプレートを修正する。
- Step 5 最終的に残った警告が許容できるものであれば、セキュリティ要求を修正する。

以上のように、実装の各段階での開発者はツールの警告を参照することで、抜け漏れのないアクセス制御機能の実装が可能となる。

表 4.9: アクセス制御機能の段階的な実装と、メトリックの変遷

Development steps	State	Trans	Warnings		
			MT3	MZ1	IN
Step 0. Original app.	46	60	0	0	0
Step 1. Add new assets	72	114	**14**	0	0
Step 2. Update PDP	72	114	14	**6**	0
Step 3. Update PEP	72	114	0	0	**145**
Step 4. Update View	72	114	0	0	12
Step 5. Suppress FP	72	114	0	0	(12)

4.2.3 既存のアプリケーションへの適用

Rails を用いた多くの Web アプリケーションが Github で公開されている。その中で、Davise と CanCan を利用しているアプリケーションを 1 3選び、Brakeman と RailroadMap による静的セキュリティテストを行った。結果を表 4.10 に示す。

Brakeman の結果では、 Rails3 の Mass Assignment の問題が多く報告された。実装に関連した XSS や SQL インジェクションの警告が若干報告されているが、フレームワークの対応を無効化しない限り、通常の実装では問題にならない。

RailrioadMap の警告では、認証の不備が多く報告されているが、これはセキュリティ要求定義ができていないため、公開ページがすべてカウントされてしまった結果である。一方認可の問題は明らかにバグである。また、Navigation のエラーが多く報告されている。これは自動生成コードを修正なしに利用しているケースが多いことが原因である。実験で用いたアプリケーションはあくまで Github で公開されたものから選んでおり、その品質レベルは不明である。ただし、この結果からわかるように、セキュリティ実装に不備が発生しやすいことは明らかである。

表 4.10: 各種 Web アプリケーションにおけるアクセス制御実装のテスト結果.

	Brakeman warnings			RailroadMap									
				Control flow Model and Policy					warnings				
Ruby on Rails Applications	MA	XSS	SQL injection	No. of states	No. of Transition .	No. of Role	No. of Obj.	PEP type	Complex policy def. (CPD)	Missing authentication (MT1+3)	Missing authorization 2 (MZ1)	Missing authorization 3 (MZ2)	Improper nav. (IN)
9balloons	0	0	0	60	92	2	2	a	1	9	1	11	8
Artdealer	0	0	0	58	74	2	2	a	2	1	0	3	0
avare	7	0	0	89	125	6	5	b, c	8	14	4	1	18
Communautaire	4	0	0	94	111	2	2	a	3	30	4	5	2
consultorio	8	0	0	94	84	2	1		1	33	2	0	0
fast-ticket	2	0	0	48	73	2	2		1	7	2	0	0
illyan	2	0	0	97	120	2	2	a, b	3	30	3	18	18
s2l	16	0	2	146	216	5	8		15	60	12	13	19
shiroipantsu	0	1	1	92	119	4	3	a	6	25	1	11	22
talks	3	0	0	135	248	3	3	b, c	4	63	9	2	74
wm-app	22	0	0	215	285	3	1	a, b	2	0	2	76	135
zmchapters	0	0	0	174	223	3	11	a, b	17	66	2	57	85
rails3-bootstrap-devise-cancan	0	0	0	46	60	2	1	b	0	0	0	0	0

4.2.4 まとめ

機能評価 (1) では、Rails を用いた Web アプリケーション開発で広く利用されているセキュリティ機能パッケージである認証機能の Devise と、アクセス制御機能の CanCan を例に、Rails の認証認可の実装からのアクセス制御ポリシーの抽出及び、ポリシー (要求) と実装との整合性の確認を行った。その結果、セキュリティ要求定義に基づいたセキュリティ機能実装の静的検証が実アプリケーションで実行可能である事が確認できた。表 4.10 に示すように、現在の Rails のアプリケーション実装ではアクセス制御実装のミスが生じやすい。しかしながら、ポリシー定義と実装コードとの整合性の自動チェックは可能であり、開発者はツールが報告するセキュリティ要求と実装の整合結果を参照することで、抜け漏れのないアクセス制御の実装が可能となる。また、表 4.9 で示すように、開発者はツールの警告を元にセキュリティ機能の実装を計画的に実施できるようになることを確認した。

4.3 機能評価 (2) テストケースの生成とテストの実施

4.3.1 テストケースの生成による XSS の検証

Rails の XSS のエスケープ機能をバイパスしているアプリケーションを例に、その検証を Cucumber によるセキュリティテストで実施する。

コードからセキュリティ検証モデルを生成する際に、XSS のエスケープを明示的にバイパスしている箇所がチェックされ、その警告を検証するためのテストケース (例 Listing 4.2) が振る舞いとデータフロー解析を元に生成される。

```
$  cucumber --tags @railroadmap_wip --format json --out cucumber.json
Using the default profile...
Rack::File headers parameter replaces cache_control after Rack 1.5.
@railroadmap_wip
Feature: XSS tests generated by RailroadMap

  # Warning          : RRMW0024
  # XSS target       : S_article#body
  # trace tag        : fRpzb4da16BAh
  # Start state      : V_article#new
  # Input            : S_article#body
  # trans            : T_V_article#new#0 render
  # trans            : T_V_article#_form#0 submit
  # trans            : T_C_article#create#0 redirect_to
  # trans            : T_C_article#show#0 render
  # Output           : V_article#show
  # Order            : 1st
  Scenario: RRMW0024 - XTP_V_article#_form_to_V_article#show_0       # features/xss_test.
      feature:15
    Given I am logged in                                             # features/
        step_definitions/user_steps.rb:53
    Given I am on the new_article page                               # features/
        step_definitions/railroadmap_steps.rb:43
    When I fill in xss_injection_msg with "fRpzb4da16BAh" for "Body" # features/
        step_definitions/railroadmap_steps.rb:76
    When I press "Create Article"                                    # features/
        step_definitions/railroadmap_steps.rb:54
    Then I should see "fRpzb4da16BAh" in raw                         # features/
        step_definitions/railroadmap_steps.rb:143
    And I should see xss_escaped_msg in raw                          # features/
        step_definitions/railroadmap_steps.rb:169
      This XSS warning is True-Positive! (MiniTest::Assertion)
      ./features/step_definitions/railroadmap_steps.rb:180:in '/^(?:|I )should see
          xss_escaped_msg in raw$/'
      features/xss_test.feature:21:in 'And I should see xss_escaped_msg in raw'

  # Warning          : RRMW0025
  # XSS target       : S_comment#body
  # trace tag        : WZT9mHALpFB2J
  # Start state      : V_comment#new => replaced
  #  replace         : needs to add steps (create new article) before new comment
  # Input            : S_comment#body
  # trans            : T_V_comment#new#0 render
  # trans            : T_V_comment#_form#0 submit
  # trans            : T_C_comment#create#0 redirect_to
  # trans            : T_C_comment#show#0 render
  # trans            : T_V_comment#show#2 link_to Back2
  # trans            : T_C_article#index#0 render
  # trans            : T_V_article#index#0 link_to Show
  # trans            : T_C_article#show#0 render
  # Output           : V_article#show
```

```
44   # Order            : 2nd
45   Scenario: RRMW0025 - XTP_V_comment#_form_to_V_article#show_0       # features/xss_test.
         feature:39
46     Given I am logged in                                            # features/
           step_definitions/user_steps.rb:53
47     Given I am on the new_article page                              # features/
           step_definitions/railroadmap_steps.rb:43
48     When I fill in "TEST1" for "Body"                               # features/
           step_definitions/railroadmap_steps.rb:66
49     When I press "Create Article"                                   # features/
           step_definitions/railroadmap_steps.rb:54
50     When I follow "Add new comment"                                 # features/
           step_definitions/railroadmap_steps.rb:58
51     When I fill in xss_injection_msg with "WZT9mHALpFB2J" for "Body" # features/
           step_definitions/railroadmap_steps.rb:76
52     When I press "Create Comment"                                   # features/
           step_definitions/railroadmap_steps.rb:54
53     When I follow "Back2"                                           # features/
           step_definitions/railroadmap_steps.rb:58
54     When I follow "Show"                                            # features/
           step_definitions/railroadmap_steps.rb:58
55     Then I should see "WZT9mHALpFB2J" in raw                        # features/
           step_definitions/railroadmap_steps.rb:143
56     And I should see xss_escaped_msg in raw                         # features/
           step_definitions/railroadmap_steps.rb:169
57       This XSS warning is True-Positive! (MiniTest::Assertion)
58       ./features/step_definitions/railroadmap_steps.rb:180:in '/^(?:|I )should see
             xss_escaped_msg in raw$/'
59 ......features/xss_test.feature:50:in 'And I should see xss_escaped_msg in raw'
60
61 Failing Scenarios:
62 cucumber features/xss_test.feature:15 # Scenario: RRMW0024 - XTP_V_article#
       _form_to_V_article#show_0
63 cucumber features/xss_test.feature:39 # Scenario: RRMW0025 - XTP_V_comment#
       _form_to_V_article#show_0
64
65 2 scenarios (2 failed)
66 17 steps (2 failed, 15 passed)
67 0m1.161s
68 Created new window in existing browser session.
69 Coverage report generated for Cucumber Features to /home/sage/Dropbox/workspace/
       securitytest4ror/rails3-bootstrap-devise-cancan-vul/coverage. 93 / 160 LOC (58.13%)
       covered.
```

Listing 4.2: Example cucumber test cases and execution log of XSS injection

このテスト結果を含めて、ダッシュボードには 図4.10 のように警告として表示される。

この作業を、４つのアプリケーションに対して適用した結果を表 4.11 に示す。最初のアプリケーションはツール評価のためのサンプルアプリケーションである。残りの３つ (intranet[10], redmine[11], vellum[12]) は Github で公開されている実際のアプリケーションを用いた。

確認方法であるが、C.1 行に各アプリケーションからセキュリティ検証モデルを抽出した際の状態遷移 (State、Transition) とデータフロー数を示す。C.1-2 行は

[10] https://github.com/fernandolopes/intranet
[11] https://github.com/redmine/redmine
[12] https://github.com/nbudin/vellum

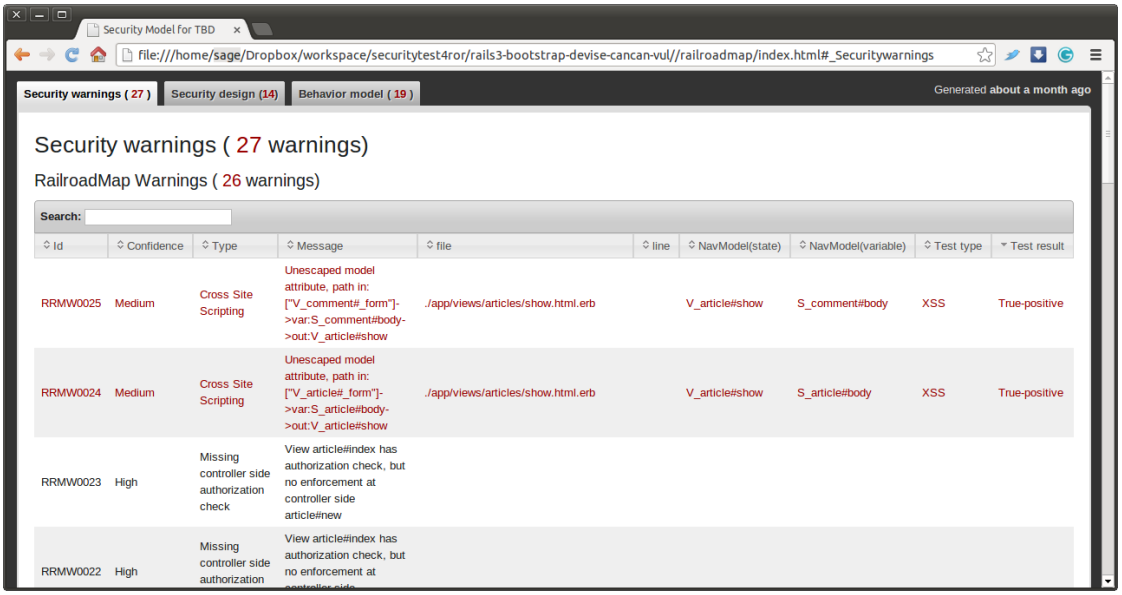

図 4.10: XSS 警告の Screenshot (version 0.1)

View のコードでエスケープされた出力箇所の数を示す。C.1-2-1 では C.1-2 で指摘された箇所から一つをサンプルとして選択し、テストケースを生成する。その結果、すべてのアプリケーションでエスケープ機能が実際に有効であることを確認できた。このサンプリングテストの目的は、アプリケーションを実際に動作させることで、フレームワーク側のエスケープ処理が実際に有効であることを確認する事である。

C.1-3 行は View のコードで RAW コマンドが用いられており、エスケープ機能が無効化された箇所の数を示す。サンプルアプリケーションでは、2 箇所 XSS 脆弱性となる RAWF コマンドが仕組まれており、ツールはこの 2 箇所を検出しテストコードを生成、テストの結果 XSS 脆弱性を持つことを確認した。

Intranet では RAW コマンドが用いられている箇所が 16 箇所あり、1 箇所は、Web 上でのリッチなテキスト編集機能を提供するプラグイン、 MCE[13]、 の利用箇所であった。ただし、MCE を使ったコンテンツの編集機能は管理者権限でしかアクセスできないページであり、前提として受け入れ可能な範囲と思われる。残りの 15 箇所については、実際には実装方法の問題で、View のコード修正を行うことで解決する問題であった。

Redmine については RAW コマンドが用いられている箇所が 9 箇所あり、テスト可能な（JSON の処理で RAW コマンドを利用している）データパスが 1 箇所あり、テストの結果、XSS 攻撃が実際に可能であった。

Vellum については RAW コマンドが用いられている箇所が 2 箇所あったが、これらは MCE を使用した箇所であり、Intranet と同様に管理者権限でしかアクセス

[13] http://www.tinymce.com/

できないページであり、前提として受け入れ可能な範囲と思われる。

現在の Rails では入力時のデータのサニタイズは行わず、出力時に一括してエスケープ処理を行っている。したがって、アプリケーション内部で HTML を含む出力メッセージを生成した場合は、RAW コマンドによりフレームワーク側のエスケープ処理を無効化する。しかしながら、このアプリケーション内部処理に脆弱性がある場合、アプリケーションの XSS 脆弱性となる。この例では、本ツールでこの問題の存在を証明することができた。

RAW コマンドを用いる場合は、XSS の脆弱性が含まれ無いことを保証することはアプリケーション側の責任となる。開発者は、危険な箇所の把握と、その実装に脆弱性がないことを、最終的にテストの形で確認することが重要である。

表 4.11: Test result

Argument tree (process)	Assessment results			
	App0 (sample app)	App1 (intranet)	App2 (redmine)	App3 (vellum)
C.1: valid model (P.1: gen. model) # of state	72	256	875	179
# of transition	120	379	954	246
# of dataflow	100	371	1911	302
C.1-2: list all escaped outputs	68	285	1754	255
C.1-2-1: select the test path (P.5: gen. test-cases)				
C.1-2-1-1: list test (P.7: run test)	1	1	1	1
C.1-2-1-1-1: test – pass	1	1	1	1
C.1-2-1-1-2: test – fail	0	0	0	0
C.1-3: list all “raw” outputs and data flows	2	16	9	2
C.1-3-1: appropriate “raw” usage	(tool test)	15 NG (fixable)	all ok	all ok
C.1-3-2: select the test path (P.5: gen. test-cases)				
C.1-3-2-1: list un-testable output	0	–	8	0
C.1-3-2-2: list testable output (P.7: run test)	2	1	1	2
C.1-3-2-1-1: no dataflow path	–		8	–
C.1-3-2-2-1: test – pass	0	0	0	0
C.1-3-2-2-2: test – fail	2(tool test)	1(MCE)	1 (fixable)	2(MCE)

4.3.2 hackety-hack.com を用いたテストケース生成機能の検証

hackety-hack.com は Rails で実装された、QA サイトであり、ソースコードが公開されている [14]。認証に Devise、認可に CanCan、テストに Cucumber を利用した一般的な Rails による Web アプリケーションである。この hackety-hack のセキュリティ保証に RailroadMap を適用し、ポリシー定義から、動的テストまでの一連の機能を評価した。

まず、作成した JSON 記述によるポリシー設定の抜粋を Listing 4.3 に示す。使用しているロールは、moderator、blog_poster、user の 3 つである。Devise に関するポリシーは自動生成されたものを用いる。実装コード上で一般にアクセスできる状態（Controller、View）は公開ページとしてポリシーを設定した。

```
{ "roles": {
    "moderator": { "level": 10, "color": "#BCA352", "description": "
        moderator"},
    "blog_poster": { "level":  5, "color": "#BCA352", "description": "
        blog_poster"},
    "user": { "level":  1, "color": "#97A750", "description": "fresh user
        " } },

  "asset_base_policies": {
  user": {
      "model_alias": { // map devise to appmodel
        "devise:registration"      : "user",
        "devise:session"           : "user",
        "devise:password"          : "user" },
      "is_authenticated": true,
      "level":            10,
      "color": "orange",
      "is_authorized":    true,
      "roles": [
        { "role": "moderator", "action": "CRUD" },
        { "role": "user",  "action": "CRU", "is_owner": true } ]},
    <snip>
   },
   "asset_discrete_policies": {
    // Devise - public
    "C_devise:session#new"         : { "is_authenticated": false, "level"
        : 0, "color": "green" },
    "C_devise:session#create"      : { "is_authenticated": false, "level"
        : 0, "color": "green" },
    "C_devise:session#destroy"     : { "is_authenticated": false, "level"
        : 0, "color": "green" },
    <snip>
    }}
```

Listing 4.3: Example requirements

[14] https://github.com/hacketyhack/hacketyhack

このポリシーを用いた静的検証の結果、ナビゲーションエラーを含むセキュリティ上の問題は見つからなかった。そこで、セキュリティ警告にもとづいたテストケース生成を行うためにフォルトインジェクションを行った。Listing 4.4 に示すように'raw' コマンドによるクロスサイトスクリプティングのエラーと、権限のチェックの無効化による Navigation error を埋め込む。埋め込んだバグに対応するツールの警告はそれぞれ、"RRMW003"、"RRMW002" となった。

```
diff --git a/app/views/questions/_list.html.haml b/app/views/questions/
    _list.html.haml
index fe52303..7a9c7c5 100644
--- a/app/views/questions/_list.html.haml
+++ b/app/views/questions/_list.html.haml
@@ -3,6 +3,7 @@
     %h3= question.answers.count
     #{question.answers.count == 1 ? "answer" : "answers"}
   .summary
+    = raw question.title
     %h2.title= link_to question.title, resource_path(question)
     .meta
       Asked by
diff --git a/app/views/questions/show.html.haml b/app/views/questions/
    show.html.haml
index b53a34b..29a84f6 100644
--- a/app/views/questions/show.html.haml
+++ b/app/views/questions/show.html.haml
@@ -3,8 +3,10 @@
   = resource.title

 - content_for :sidebar do
-  - if can? :update, resource
-    = link_to 'Edit', edit_resource_path, :class => "edit btn"
+  NAV ERROR
+  = link_to 'Edit', edit_resource_path, :class => "edit btn"
+  -# if can? :update, resource
+  -#  = link_to 'Edit', edit_resource_path, :class => "edit btn"
   - if can? :destroy, resource
     = link_to 'Delete', resource_path, :class => "delete btn", :confirm
         => 'Are you sure?', :method => :delete
```

Listing 4.4: Example requirements

次に、各種のセキュリティ機能と、セキュリティ警告を確認するためのテストケースの生成を行う。テストケース生成の制御は "testplan.json" ファイルで行う。Listing 4.5 がフォルトインジェクションされた hackety-hack に対するテストプランの抜粋である。前半では CSRF 対策や、アクセス制御に関するコマンドのテストケース生成のための設定となる。

```
{
  // CSRF
```

```
  "protect_from_forgery": {
    "type": "csrf",
    "testcase_type": "Modification of existing testcase",
    "testcase": {
      "init_steps": [
        "Given I'm logged in as someone who can post to the blog",
        "When I visit the blog admin page"
        //"And I fill out the new blog form"
      ],
      "target_step": "And I press \u0022Create Blog post\u0022",
      "success_step": "Then I should see that my post has been created"}
<snip>
  // Devise
  "authenticate_user!": {
    "type": "authentication",
    "testcase_type": "automatic",
    "target_path":  "questions",
    "success_text": "Have A Question?",
    "error_text":   "You need to sign in or sign up before continuing."
  },
  // CanCan
  "load_and_authorize_resource": {
    "type": "global_authorization",
    "testcase_type": "automatic",
    "strong_privilege_user": "blog_poster",
    "weak_privilege_user"  : "user",
    "strong_privilege_path": "blog admin page",
    "success_text": "New Post",
    "error_text":   "You are not authorized to access this page"
  },
  // CanCan View can
  "can?": {
    "type": "conditional_authorization",
    "testcase_type": "Use existing testcase",
    "testcases": {
      "navigation and authorization - OK": [
        "Given questions exist",
        "Given a logged in moderator",
        "When I visit the questions page",
        "When I click the first question link",
        "# Then show me the page",
        "Then I should see \u0022Edit\u0022"
      ],
      "navigation and authorization - NG": [
        "Given questions exist",
        "Given a logged in user",
        "When I visit the questions page",
        "When I click the first question link",
        "# Then show me the page",
        "Then I should not see \u0022Edit\u0022"
      ]
    }
  },
```

```
56
57 <snip>
58   // Make sure, this is a Nav. error
59   "RRMW0002": {
60     "type": "Missing view side authorization check",
61     "testcase_type": "Use existing testcase",
62     "testcases": {
63       "replay navigation error": [
64         "Given questions exist",
65         "Given user exist",
66         "When I visit the questions page",
67         "When I click the first question link",
68         "When I click \u0022Edit\u0022",
69         "And I should see \u0022Sign In\u0022"
70       ],
71       "TO BE": [
72         "Given questions exist",
73         "Given user exist",
74         "When I visit the questions page",
75         "When I click the first question link",
76         "And I should not see \u0022Edit\u0022"
77       ]
78     }
79   },
80   "RRMW0003": { // XSS Type 1: Reflected XSS (or Non-Persistent)
81     "type": "Cross Site Scripting",
82     "subtype": "Type 1: Reflected XSS",
83     "testcase_type": "Modification of existing testcase",
84     "testcase": {
85       "init_steps": [
86         "Given I am a user that has created a question",
87         "When I visit the new question page"
88         //"And I fill out the new blog form"
89       ],
90       "target_steps": [
91         "And I fill in \u0022Title\u0022 with",
92         "And I fill in \u0022Description\u0022 with"
93       ],
94       "post_step": "And I press \u0022Ask Everyone\u0022"}}}
```

Listing 4.5: Example test plan

CSRF のテストプランでは、アプリケーション上の任意の POST(Submit ボタン) を行うページの正常なアクセスを Cucumber のテストシナリオで与える。これに対応して生成されたテストケースを Listing 4.6 に示す。与えられた正常なテストシナリオとそれを元に、hidden_field に対して改ざんを加えるテストシナリオの 2 つが生成される。後者の場合は、正常であれば CSRF のチェックにより、ホームページへリダイレクトされる。

テストケースの生成については、個別のコマンドや警告に応じた設定方法を用いている。Device、CanCan などの既知のパッケージを用いた場合には、一般的

なテストシナリオが利用できるため、開発者はテストケース生成の場所や、テストで利用する文字列等を設定する。CSRF や XSS などの不正な値の入力については、通常の動作シナリオから派生して、攻撃シナリオを生成させている。

```
Feature: CWE-352: Cross-Site Request Forgery (CSRF)
  The web application does not, or can not, sufficiently verify
  whether a well-formed, valid, consistent request was intentionally
  provided by the user who submitted the request.

  Scenario: RRMT0005[0]
    normal access (existing testcase)

    Given I'm logged in as someone who can post to the blog
    When I visit the blog admin page
    And I press "Create Blog post"
    Then I should see that my post has been created

  Scenario: RRMT0005[1]
    tampered hidden_field (existing testcase)

    Given I'm logged in as someone who can post to the blog
    When I visit the blog admin page
    And I press "Create Blog post" with tampered authenticity_token
    Then I should see that you need to sign in or sign up before
        continuing.
```

Listing 4.6: Example requirements

4.3.3 まとめ

機能評価 (2) では、Rails の Web アプリケーションの実装コードを静的解析し、セキュリティ上問題となる箇所を特定し、その箇所に対するセキュリティテストケースを生成することで、問題箇所が実際に脆弱性を持つか確認する手法について評価した。

この実験ではテストフレームワークの Cucumber を用いたセキュリティテストケースの自動生成が可能であることを確認した。表 4.11 に示すように、フレームワークのエスケープ機能の有効性の確認、静的検証で指摘した問題箇所の、(動的) テストによる脆弱性の判定が可能となることを確認した。

また、hackety-hack.com を用いた実験では、CSRF や XSS などのような実装に起因するセキュリティ問題に追加して、設計に関係するアクセス制御ポリシーの作成からその実装コードの静的検証および動的な機能テストまでを一貫してツールで管理する手法について有効性を確認した。

4.4 機能評価 (3) セキュリティテストカバレッジの計測

セキュリティテストカバレッジの計測の実験として、Foreman[15] に提案手法を適用して、メトリックスの変化を計測した。Foreman は仮想環境のライフサイクル管理ツールであり、プロセス管理で広く利用されている。また、脆弱性管理が実施されており [16]、ソースコードとともに提供されるテストケースが充実している。そのため、セキュリティテストカバレッジの計測の例として選択した。

次の 4 つのフェーズで提案手法を Foreman に対して適用し、セキュリティテストカバレッジの向上が可能か実験を行う。

Phase 0 事前準備、アプリケーションのセキュリティ機能の理解

Phase 1 ツールの初期化

Phase 2 セキュリティテストカバレッジの検証、及びツールとテストケースの修正

Phase 3 テストケースの追加によるセキュリティテストカバレッジの向上

次に、各フェーズでの作業を、アクセス制御のテストと、SINK カバレッジの 2 つの観点でまとめる。

4.4.1 事前準備 (Phase 0)

Foreman が提供する、各種の資料を調べ、そのセキュリティ機能について概要を理解する。Foreman は RBAC によるアクセス制御を独自実装でサポートする。したがって、Foreman が提供するセキュリティ機能のコマンド抽象化ライブラリを定義する必要がある。また、Foremen が該当する CVE が 24 個報告されている [17]。その際に、セキュリティテストケースが追加されていることが、レポジトリのコミットから確認できた。また、Rails 標準テスト・フレームワークを用いており、SimpleCov によるカバレッジ計測を行える。

4.4.2 ツールの初期化 (Phase 1)

RailroadMap の初期化を行う。また、未定義のコマンドに対応して、コマンド抽象化ライブラリのテンプレートを生成する。初期の制御フローモデルは、 1392 の状態と 1154 の確定した遷移と、748 の未確定の遷移を持つ。未確定の遷移が多い理由は、アプリケーションが 2 つの異なるバージョンの JSON API を持ち、それを切り替えて使うためである。2735 個のテストケースを持ち、内訳は ユニッ

[15] http://theforeman.org/
[16] http://theforeman.org/security.html
[17] 期間は 2012 − 2014

トテストが 1295、機能テストが 1207、統合テストが 128 であった。この段階で未定義なコマンドは 552 個である。

Foreman はアクセス制御リストをデータベースで管理する。テストの際に使用するアクセス制御リストは Rails の fixtures [18] の形で用意される。したがって、この fixtures 定義からアクセス制御リストを抽出できるようにした。抽出したアクセス制御リストとテストとの関係を表 4.12 に示す。テストケースは次の 4 つに分類して表に示す。

T 成功テスト (Positive test)

F 失敗テスト (Negative test)

B T と F の両方

x 未分類

これらはテストコードの Assert 文から判定する。この段階での、アセット (制御フローの Controller state、 ACL の Object に相当) のテストカバレッジは 39.29%、ロール (ACL の Subject に相当) のテストカバレッジは 90.91% であった。

Foreman で用いられているセキュリティに関係するコマンド (SINK) の一覧と、そのテストカバレッジを表 4.13 に示す。列はそれぞれ、コマンド名、コマンドのタイプ、コマンド抽象化ライブラリ、コード中の利用箇所の数、カバレッジとの対応、SC と RC での集計、対応するテストケース、を示す。RC は 9 つ検出された。SympleCov のカバレッジ集計は View のテンプレートはサポートしないため、View 内の SINK はカバレッジが不明である。この段階での SINK カバレッジはユニットテストの SC カバレッジが 66.67%、機能テストの SC カバレッジ 88.89%、高リスクの RC カバレッジが 該当なし、中リスクの RC カバレッジが 57.32%、低リスクの RC カバレッジが 5.85%となった。

4.4.3 カバレッジの確認とツールと実装の修正 (Phase 2)

未定義なコマンドをアプリケーションのコマンド抽象化ライブラリに登録する。その際に、セキュリティコマンドについては定義（SC,RC の分類、セキュリティ機能の分類）を行う。その結果、168 のテストケースがセキュリティコマンドに関するものとして検知された。

アセットのテストについて、実装コード、テストケース、ポリシー定義を再確認した。その結果、 4 つのアセットが、ポリシー上の名前と、実装の名前に一致しない（Alias されている）事がわかった。このような、Alias 機能に対応するようにツールの機能を拡張することで、アセットのテストカバレッジは 75.00% に向上した。

[18] テスト実行時に用いるデータベースの初期値のセット

SINK については、13 の Foreman が提供するセキュリティコマンド (SC) をコマンド抽象化ライブラリに追加することで、SC のカバレッジが向上した。この段階で SINK カバレッジは、ユニットテストの SC カバレッジが 81.82%、機能テストの SC カバレッジ 95.45%と向上し、高リスクの RC カバレッジが 該当なし、中リスクの RC カバレッジが 57.32%、低リスクの RC カバレッジが 5.85%と変化した。RC については、View のコードに存在する物が多く、これらについてはテストカバレッジが計測できない。CSRF 対策については、テストコードは存在するが、テストで CSRF 特有のコマンドを使用しないため、検知できない。そこで、テストケースの記述から対応を推測できるように、キーワードでの検査機能を追加した。テストケース名に CSRF の CWE エントリである “CWE-352” を記載することで、そのテストケースが CSRF のテストであることを認識する。

4.4.4 テストケース追加によるカバレッジの向上 (Phase 3)

アクセス制御に関して、8 つのテストケースを追加した。また、コード上では削除されたが、テストケースのアクセス制御定義に残っていた不要なアセットを削除した。その結果、アクセス制御に関するテストカバレッジは 100%となった。

SINK に関しては、フレームワークの SC について 3 つのユニットテストを追加したが、カバレッジの変更はない。

表 4.12: Foreman のアクセス制御リストとテストカバレッジ (→:phase2, ⇒:phase3)

Asset (class)	cov	Manager	Default user	Anonymous user	Viewer	View host	Create host	Edit host	Destroy host	CRUD host	View comp. res.	Edit part. table
architectures	[x]	[T] CRUD	R	[F]	[B] R							
authenticators	[→x]	CRUD	R	[→F]	[→B] R							
environments	[x]	CRUD	R	[x]	[x] R							
external_variables	[→x]	CRUD	R	[→x]	[→x] R							
domains	[x]	CRUD	R	[F]	[B] R							
globals	[→x]	CRUD	R	[→F]	[→B] R							
hostgroups	[x]	[B] CRUD	[T] R		[T] R							
hosts	[x]	[B] CRUD	[T] R		[T] R	[T] R	[⇒B] C	[B] U	[B] D	[B] CRUD		
media	[→x]	CRUD	R	[→F]	[→B] R							
models	[→x]	CRUD	R	[→F]	[→B] R							
operatingsystems	[→x]	CRUD	R	[→F]	[→B] R							
ptables	[x]	CRUD	R	[F]	[B] R							[x] CRUD
puppetclasses	[→x]	CRUD	R	[→F]	[→B] R							
usergroups	[→x]	CRUD		[→F]	[→B] R							
users	[x]	[B] CRUD		[F]	[B] R							
settings	[⇒x]	[⇒x] A	A		A							
dashboard	[⇒x]	[⇒T] A	A		A							
statistics	[→x]	R	R		[→T] R							
reports	[x]	RD	R		[T] R							
facts	[→x]	R	R		[→T] R							
audit_logs	[→x]	R			[→T] R							
locations	[⇒x]	[⇒T] CRUD			R							
organizations	[⇒x]	[⇒T] CRUD			R							
classes	[⇒x]	[⇒T] U										
realms	[x]	CRUD	R	[F]	[B] R							
tasks	[⇒x]			[⇒B] R								
compute_resources	[x]										[x] CRUD	
→templates	[x]	[x] →CRUD										
Coverage (phase 1)[%]	39.29	16.00	10.50	50.00	40.00	100.0	0.00	100.0	100.0	100.0	100.0	100.0
Coverage (phase 2)[%]	75.00	16.00	10.50	50.00	80.00	100.0	0.00	100.0	100.0	100.0	100.0	100.0
Coverage (phase 3)[%]	100.0	36.00	10.50	100.0	80.00	100.0	100.0	100.0	100.0	100.0	100.0	100.0

4.4.5 まとめ

以上、Foreman での RailroadMap の使用環境を整える作業を 4 つのフェーズに分割し、その作業の中で変化するメトリックスを記録した。このメトリックスの変化を図 4.11 に示す。セキュリティ機能に関するアクセス制御と SC の SINK については、Foreman のテストケースは高いテストカバレッジを持つことがわかる。また、表 4.12 や表 4.13 のような形式でのカバレッジ情報の一覧表示は、セキュリティの現状の把握と改善に有効である。一方、RC については十分と言えないが、これは RC が View のテンプレートコードに多く存在する事が理由の一つである。

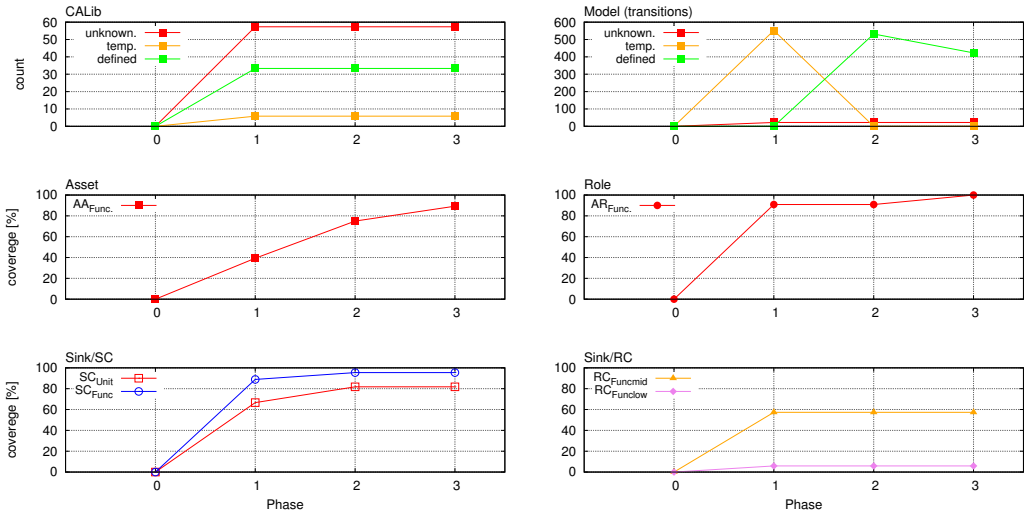

図 4.11: メトリックス駆動によるセキュリティ保証

表 4.13: Foreman の SINK コマンドとテストカバレッジ (→:phase2, ⇒:phase3)

Command	type				CALib	num. of sink						SimpleCov				SC cov.			RC cov. (score)		Test case			
	SC	RC	CF	DF		all	M	C	V	S	skip	all	U	F	I	U	F	I	(4-3)	(2-1)	all	U	F	I
protect_from_forgery	X	-	-	-	rails	2	0	2	0	0	0	6	2	2	2	1/1	1/1	1/1			0→2	0	0→2	0
escapeHTML	X	-	-	-	ruby	4	0	4	0	0	0	234	0	221	13	1/1	1/1	1/1			0⇒1	0⇒1	0	0
html_escape	X	-	-	X	rails	1	0	1	0	0	0	12	0	12	0	0/1	1/1	0/1			0⇒1	0⇒1	0	0
h	X	-	-	X	rails	39	0	0	39	0	0	NA	NA	NA	NA	0/1	0/1	0/1			0⇒1	0⇒1	0	0
before_filter	X	-	-	-	rails	197	0	197	0	0	0	1886	232	1146	508	1/1	1/1	1/1			0	0	0	0
skip_before_filter	X	-	-	-	rails	11	0	11	0	0	0	36	12	12	12	1/1	1/1	1/1			0	0	0	0
validates	X	-	-	-	rails	165	165	0	0	0	0	22687	12249	9618	820	1/1	1/1	1/1			2	2	0	0
before_save	X	-	-	-	rails	18	18	0	0	0	0	1887	1316	547	24	1/1	1/1	1/1			0	0	0	0
attr_accessible	X	-	-	-	rails	21	21	0	0	0	0	63	21	21	21	1/1	1/1	1/1			0	0	0	0
find	-	X	-	-	rails	66	4	62	0	0	0	1039	478	548	13				33/63	0/0	(115)	22	32	61
where	-	X	-	-	rails	17	9	8	0	0	0	22046	10170	11667	209				23/26	0/0	(39)	24	5	0
send_file	-	X	-	-	rails	1	0	1	0	0	0	EX	EX	EX	EX				0/1	0/0	0	0	0	0
constantize	-	X	-	-	rails	2	0	2	0	0	0	360	0	154	206				1/1	0/0	0	0	0	0
redirect_to	-	X	X	-	rails	34	0	34	0	0	0	379	0	313	66				0/1	11/31	1	1	0	0
raw	-	X	-	X	rails	21	0	0	21	0	0	NA	NA	NA	NA				0/0	0/21	0	0	0	0
html_safe	-	X	-	X	rails	8	0	0	8	0	0	NA	NA	NA	NA				0/0	0/6	0	0	0	0
link_to	-	X	X	-	rails	94	0	0	94	0	0	NA	NA	NA	NA				0/0	0/94	0	0	0	0
content_tag	-	X	-	-	rails	36	0	0	36	0	0	NA	NA	NA	NA				0/0	0/36	0	0	0	0
→authenticate	→X	-	-	-	app	12	9	2	0	1	0	1201	18	1134	49	0/1	1/1	1/1			→7	→7	0	0
→require_login	→X	-	-	-	app	7	0	4	0	1	2	32	4	24	4	1/1	1/1	1/1			→4	0	→4	0
→authenticated?	→X	-	-	-	app	2	0	2	0	0	0	98	0	14	84	1/1	1/1	1/1			→4	0	→4	0
→is_admin?	→X	-	-	-	app	2	0	1	0	1	0	18	1	16	1	0/1	1/1	1/1			0	0	0	0
→require_admin	→X	-	-	-	app	8	0	8	0	0	0	24	8	8	8	1/1	1/1	1/1			0	0	0	0
→authorize	→X	-	-	-	app	12	0	8	0	0	4	27	9	9	9	1/1	1/1	1/1			→2	→2	0	0
→authorized	→X	-	-	-	app	15	1	13	0	1	0	1547	37	1327	183	1/1	1/1	1/1			→6	→4	→2	0
→require_ssl	→X	-	-	-	app	2	0	2	0	0	0	6	2	2	2	1/1	1/1	1/1			0	0	0	0
→encrypts	→X	-	-	-	app	2	2	0	0	0	0	6	2	2	2	1/1	1/1	1/1			0	0	0	0
→encrypt_setters	→X	-	-	-	app	2	2	0	0	0	0	6	2	2	2	1/1	1/1	1/1			0	0	0	0
→encrypt_field	→X	-	-	-	app	1	1	0	0	0	0	3	1	1	1	1/1	1/1	1/1			→3	→3	0	0
→encrypts?	→X	-	-	-	app	1	1	0	0	0	0	3	1	1	1	1/1	1/1	1/1			→1	→1	0	0
→parse	→X	-	-	-	app	4	3	1	0	0	0	27	12	15	0	1/1	1/1	0/1			→7	→7	0	0

SC:security command, RC:Risky, CF:Control flow, DF: Data flow
M:Model, C:Controller, V:View, S: support/helper/library, skip(disabled)
U:Unit, F:Functional, I:Integration

4.5 機能評価 (4) セキュリティ知識との連携

この節では、CWE,CAPEC を用いたセキュリティ知識と、アプリケーション実装コードの対応付けの実験を行う。まず、CWE,CAPEC を 3.4 節で示した方法で、Rails に関する項目に絞る。次に、Rails のサンプル・アプリケーション sample_app_3rd_edition[19] と、Rails の Broken Web Application (BWA) である Railsgoat [20] に適用し、実装コードと脆弱性情報との関連について調べた。

4.5.1 Rails に対応した CWE, CAPEC のサブグラフの作成

まず、CWE,CAPEC のエントリを Rails に関係するものに絞り、コマンド抽象化ライブラリに登録されている、SC、RC と対応づける。CWE については Version2.8 を、CAPEC については Version 2.7 を用いた。

4.5.1.1 CWE,CAPEC の読み込み

CWE,CAPEC については、公開されている最新の情報（XML 形式）をダウンロードした後、ツールに読み込ませて、項目間の依存関係を元に有向グラフを作成する。表 4.15 に、以下の作業に伴う、CWE、CAPEC の選択数の変化を示す。

4.5.1.2 アプリケーション・ドメイン選択

Web アプリケーションのドメイン選択には下記を用いた。その結果、CWE の数は 1003 から 473 に半減した。CAPEC の数についても、463 から 267 へと削減された。

- CWE-809: Weaknesses in OWASP Top Ten (2010)
- CWE-928: Weaknesses in OWASP Top Ten (2013)
- CWE-442: Web Problem

4.5.1.3 アプリケーション・フレームワーク選択

ここでは J2EE や.NET のような Rails 以外のフレームワークや、PHP などの Ruby 以外の言語固有の問題を除外する。その結果、CWE の数は 473 から 383 に半減した。CAPEC の数についても、267 から 191 へと削減された。

[19] https://github.com/mhartl/sample_app_3rd_edition
[20] https://github.com/OWASP/railsgoat

4.5.1.4 コマンド抽象化ライブラリの更新

コマンド抽象化ライブラリの SC,RC と、選択された CWE,CAPEC とを対応付ける。CVE に登録された Rails に関する脆弱性で、実装コードの問題箇所と CWE の関係が特定できる場合は、その関係を登録する。この段階で、分類対象となるコマンドを表 4.14 に示す。対象となるコマンドの数は、SC で 33 個、RC で 15 個であり、それはツールの開発者で十分に分類可能な数である。また、Rails の実装コードを関連付けられた CWE は 31、CAPEC は 8 となった。

Rails などに固有な Mass Assignment の脆弱性については、CWE-915 が該当するが、ドメイン選択、フレームワーク選択で選択した CWE には含まれていない。そのため、選択でもれた CWE については個別に追加する。

表 4.14: コマンド抽象化ライブラリのコマンド数と SC,RC 数

Lib.	Total	SC	RC	SC example	RC example
Ruby	29	1	5	escapeHTML	eval, system
Rails	145	15	10	html_escape	html_safe, raw, redirect_to
Devise	44	9	0	sign_in, sign_out, sign_up	–
CanCan	9	8	0	authorize!, can, cannot	–

こうして得られた CWE、CAPEC とコマンド抽象化ライブラリの SC,RC との関係をグラフで表示すると、図 4.12 のようになる。ここで、SC を緑の楕円、RC を赤の楕円で示す。CWE のタイプを、View は白色の四角、Category をシアン色の四角、 Weakness をピンク色の四角で示す。枠線の色は、関連するコマンドが SC の場合は緑、RC の場合は赤、両方の場合は青となる。ノード間の接続線も同様の配色としている。CAPEC についても、Category をシアン色の四角、 Attack Pattern をピンク色の四角で示す。アクセス制御に関する SC と、クロスサイトスクリプティングに関係する、SC と RC が多い事がわかる。

比較のため、図 4.13 に Rails のコマンドと対応できなかった CWE、CAPEC も含めた関係図を示す。関連がなかった CWE,CAPEC はグレーで表示する。CWE には 400 近いエントリが残っており、その全体からみると、Rails のコード実装と対応する CWE の数、31 は全体の 1 割弱であった。

表 4.15: CWE,CAPEC の選択数の推移

		Reduction			Rails			sample_app_3rd_edition			Railsgoat		
	Type	ALL	Web app.	Rails	CALib	SC	RC	CALib	SC	RC	CALib	SC	RC
CWE	View	32	2	2	2	2	2	2	2	2	2	2	2
	Category	244	47	36	15	12	9	13	5	9	17	9	16
	Weakness	719	418	342	13	8	10	10	5	0	21	15	10
	C. E.	8	7	5	1	1	0	0	0	0	0	0	0
	total	1003	474	383	31	23	21	25	12	11	40	26	28
CAPEC	View	8	0	0	0	0	0	0	0	0	0	0	0
	Category	73	20	16	1	1	0	2	2	0	1	1	0
	Attack P.	463	267	191	8	5	5	9	6	3	10	7	3

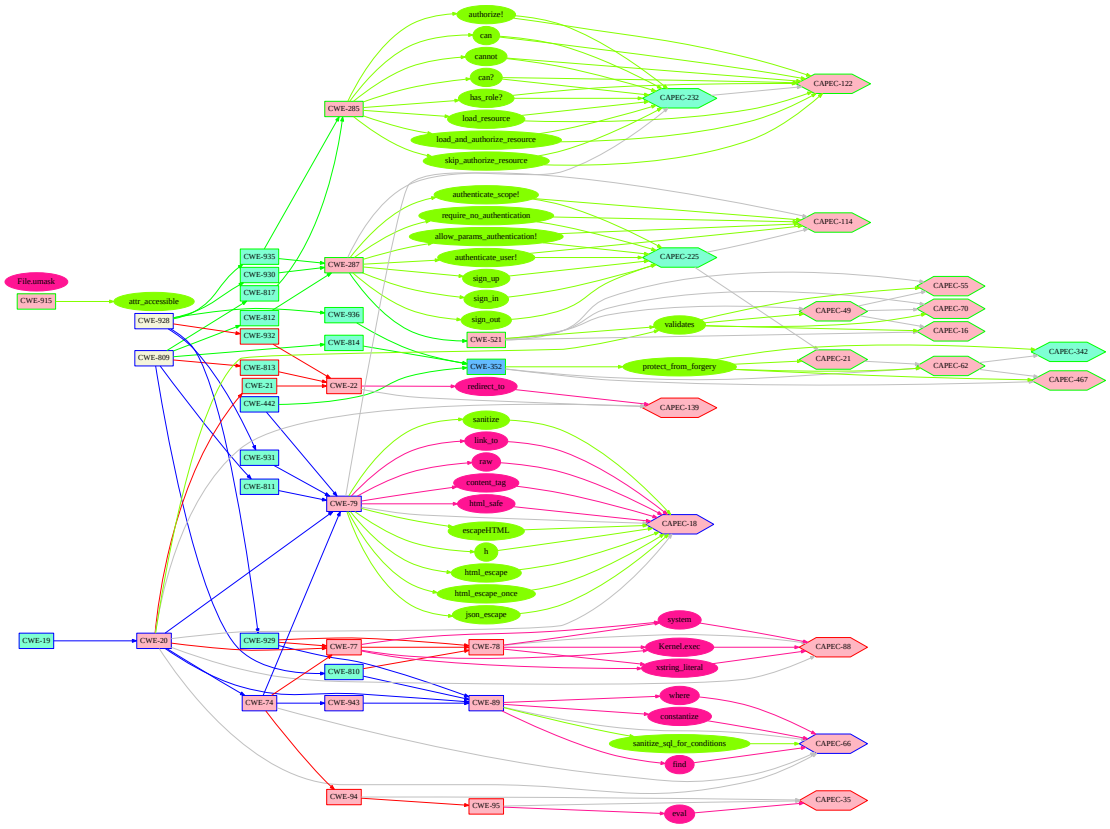

図 4.12: Rails に関連する CWE と CAPEC のグラフ表現

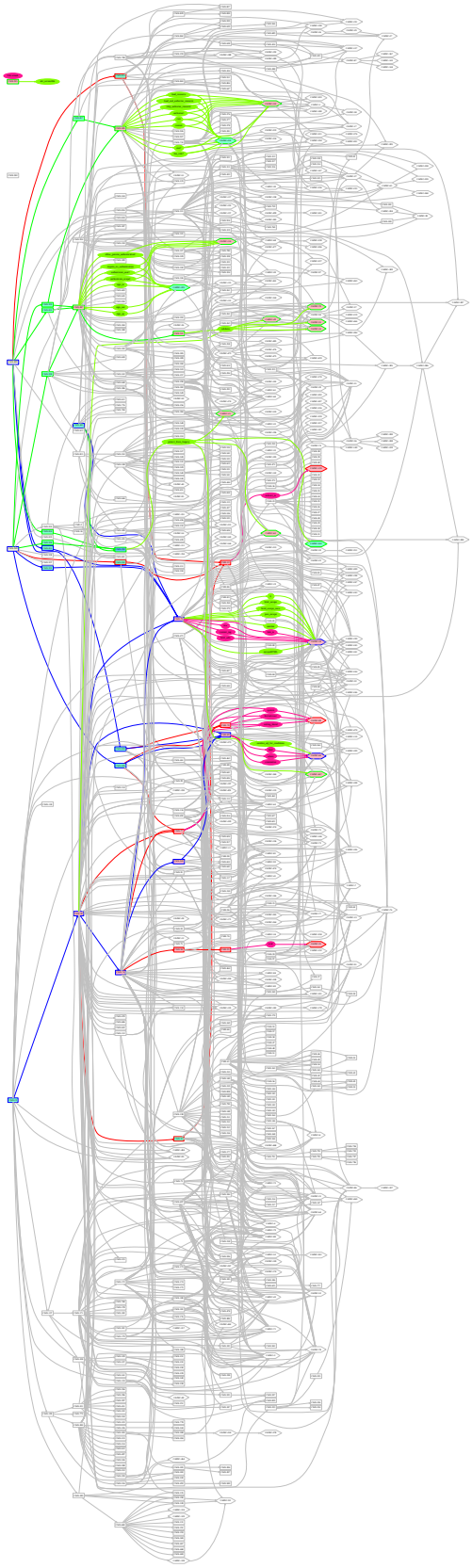

図 4.13: Rails に関連する CWE と CAPEC のグラフ表現（SC,RC と関連しない CWE、CAPEC を表示）

4.5.2 sample_app_3rd_edition への適用

次に、実際の Web アプリケーションとして、sample_app_3rd_edition の実装コードと、CWE,CAPEC の関係を調べる。sample_app_3rd_edition は Rails の入門書籍[21] で開発されるソーシャル・ネットワークの Web アプリケーションであり、テストケースも付随することから、実験対象として選択した。

sample_app_3rd_edition のセキュリティ要求、CWE、コマンド、テストカバレッジ の対応を表 4.16 にまとめる。実装コードに現れる SC,RC は 10 個であり、付属するテストケースもこれらをテストしていることを確認した。CWE,CAPEC との対応は、図 4.14 にグラフの形でも示す。CWE-915 の Mass Assignment については、CAPEC に攻撃パターンが登録されていないが、その他については CWE,CAPEC との対応が確認できた。

表 4.16: sample_app_3rd_edition: セキュリティ要求、CWE、コマンド、テストカバレッジ の対応

Security requirement	CWE	cmd, type	CALib	CAPEC	location			test coverage		
					M	C	V	0	1	n
Authentication	287	SC	logged_in_user	255,224	0	4	0	0	4	0
	287	SC	log_in	255,224	0	3	0	0	2	14
Authorization	285	SC	admin_user	232,122	0	2	0	0	2	0
Input Validation	521,20	SC	validate	49,16,55,70	7	0	0	0	7	0
	915	SC	user_params		0	2	0	0	2	0
	915	SC	micropost_params		0	1	0	0	1	0
	22	RC	redirect_to	139	0	17	0	1	8	8
	22	RC	redirect_back_or	139	0	1	0	0	0	1
Escaping output	79	RC	link_to	18	0	0	26	-	-	-
	79	RC	raw	18	0	0	1	-	-	-

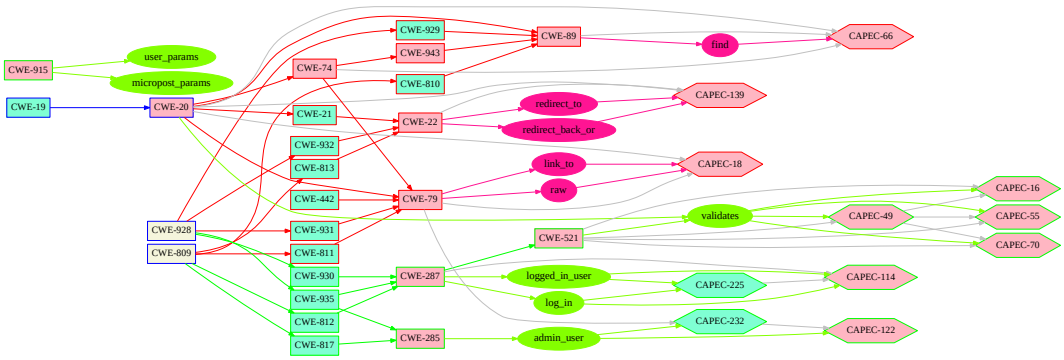

図 4.14: sample_app_3rd_edition の CWE,CAPEC,SC,RC グラフ表示

[21] RUBY ON RAILS TUTORIAL (3RD ED.), https://www.railstutorial.org/

4.5.3 Railsgoat への適用

Railsgoat は OWASP が開発した Broken Web Application であり、OWASP Top 10 (CWE-928, 938) にリストされた脆弱性を持つ。そのため、既知の脆弱性を含む Web アプリケーションの実装コードに、本手法を適用することで、CWE,CAPEC の情報が実装コードの実際の脆弱性と関連付けられるかを確認する。表 4.17 に Railsgoat に含まれる脆弱性とコマンド (SC,RC)、CWE と CAPEC、コード上の場所について整理する。Railgoat に含まれる脆弱性のうち、設定に関する脆弱性 (A3.1、A5、A9) は実装 MVC コードとは無関係なため、対象から除くが、それ以外の脆弱性についてはコマンドと対応する事がわかる。カッコで囲んだコマンドについては、Railsgoat の提供するコマンド内部で使用されているコマンドである。また、CWE,CAPEC,SC,RC の関係を図 4.15 に示す。

表 4.17: Security requirements and commands(Railsgoat)

Flaws	command		CWE	CAPEC	location		
	RC	SC			M	C	V
A1.1 SQL Injection Concatentation (IDOR)	find	–	929, 89	66			
A1.2 SQL Injection Interpolation	where	–	929, 89	66			
A1.3 Command Injection	make_backup		929, 77,78	88	1	0	0
	(system)	(FileUtils.cp)					
A2.1 Credential Enumeration	Exception.message	–	930,319,522	169	0	5	0
A2.2 Lack of Password Complexity	validates	validates	930,287,521,20	112	1	0	0
A2.3 Insecure Compare and Timing Attacks	authenticate		930,208	224	0	6	0
	(Digest::MD5.hexdigest)	(Rack::Utils.secure_compare)					
A2.4 Lack of HttpOnly Flag (config)	–	–	930,82	–	–	–	–
A3.1 Cross Site Scripting	html_safe	h	931,79	18	0	0	8
A3.2 Cross Site Scripting DOM Based (JS)	*document.write*	*hoganEscape*	931,79	19	–	–	–
A4 Insecure Direct Object Reference (IDOR)	User.find_by_user_id		932,99	–			
A5 Security Misconfig Modification (config)	–	–	933, 16	–	–	–	–
A5 Security Misconfig JSON Escaping (config)	–	–	933, 16	–	–	–	–
A6 Sensitive Data Exposure Insecure Password Storage	authenticate		934,326	112	0	6	0
	hash_password		934,326	112	2	0	0
	(Digest::MD5.hexdigest)	(BCrypt::Engine.hash_secret)					
A6 Sensitive Data Exposure Cleartext Storage SSNs	encrypt_ssn		934,312	169	1	0	0
	decrypt_ssn		934,312	169	1	0	0
		OpenSSL::Cipher::Cipher.new					
A6 Sensitive Data Exposure Model Attributes Exposure	as_json	as_json	934,319	169			
A7 Missing Function Level Access Control	admin_param	before_filter, administrative	935,285	1			
A8 Cross Site Request Forgery		protect_from_forgery	936,352	21,62,342,467	0	*1*	0
A9 Using Components with Known Vulnerabilities	–	–	937	–	–	–	–
A10 Unvalidated Redirects and Forwards	redirect_to	URI.parse	938,601,22	139	0	20	0
Extras Mass-Assignment (Rails3)	attr_accessible	attr_accessible	915	–	8	0	0
Extras Logic Flaws Broken Regular Expression		identify_user	185	–			
Extras Logic Flaws Insecure Encryption Reuse		encrypt_sensitive_value	323	–			
Extras Metaprogramming	constantize	–	89	66			

4.5.4 まとめ

以上の実験から、Rails の実装コードに含まれる、SC,RC と CWE,CAPEC とを対応づけられることが十分に可能である事がわかった。

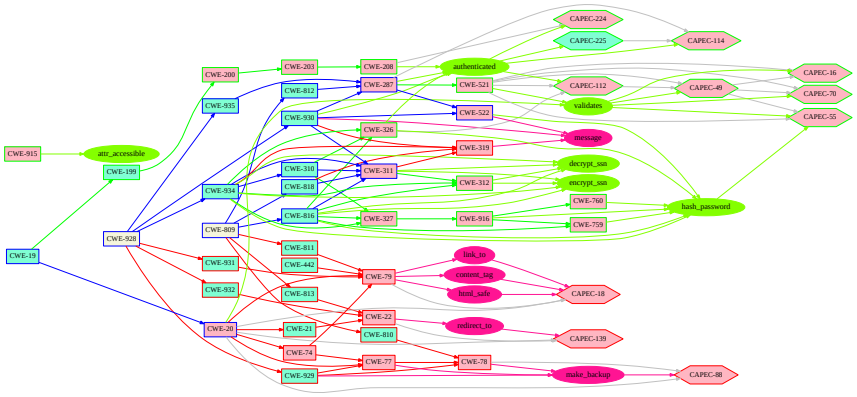

図 4.15: Railsgoat の CWE,CAPEC,SC,RC グラフ表示

4.6 既存のセキュリティテスト手法との比較

4.6.1 評価用 Web アプリケーション

脆弱性検査の演習及び、脆弱性検査ツールの評価の目的で、表 4.18 に示すように、様々な脆弱性を仕込んだ評価用の Web アプリケーション (BWA: Broken Web Application) が公開されている。

表 4.18: 評価用脆弱 Web アプリケーション

名前	言語	含まれる脆弱性
BadStore.net		
moth (2009)		
Hacme Bank (2006) [22]	.NET	
Hacme Casino(2006)	Rails	
WebGoat [23]	J2EE or .NET	
SiteGenerator [24]	.NET	
WackoPicko [30] [25]	PHP	
Vicnum [26]	Perl/PHP	
DVWA [27]	PHP	
Gruyere[28]	Python	
bwa_cyclone_transfers (2012-)	Rails	MA, XSS, SQL injection, file upload weakness session management
Rails Goat (2013-)	Rails	OWASP Top 10

様々な言語、フレームワークで実装されているが、フレームワークとして Rails を用いたものは、Hacme Casino [29] 、bwa_cyclone_transfers [30] 、Rails Goat [31] の 3 つである。そこで、最新の Rails Goat を用いて、RailroadMap の機能評価と、その他のセキュリティテスト・ツールとの比較を次節行う。

4.6.2 RailsGoat を用いたセキュリティテストツールの比較

4.6.2.1 RailsGoat に含まれる脆弱性

RailsGoat は表 4.19 で示す OWASP Top 10 の脆弱性を含む、Broken Web Application(BWA) である。個別の脆弱性の詳細は、4.6.2.4 節の評価結果のまとめで述べる。

[29] http://www.mcafee.com/us/downloads/free-tools/hacme-casino.aspx
[30] https://github.com/fridaygoldsmith/bwa_cyclone_transfers
[31] https://www.owasp.org/index.php/OWASP_Rails_Goat_Project

表 4.19: RailroadMap の変更履歴とロードマップ

A1	INJECTION
A2	BROKEN AUTH
A3	XSS
A4	INSECURE DOR
A5	MISCONFIG
A6	EXPOSURE
A7	ACCESS
A8	CSRF
A9	COMPONENTS
A10	REDIRECTS
EXTRAS	

4.6.2.2 セキュリティテストツールの準備

比較に用いるツールの選定にあたっては、Rails 開発者が利用しやすいものを選定した。表 4.20 にその一覧を示す。

表 4.20: RailroadMap の変更履歴とロードマップ

ツール名	タイプ	利用 (ライセンス)
bundler-audit	静的テスト	無償
brakeman	静的テスト	無償
codesake-dawn	静的テスト	無償
W3AF	脆弱性スキャナ	無償
skipfish	脆弱性スキャナ	無償 (Apache 2.0)
ZAF	脆弱性スキャナ	無償
railroadmap	提案手法	無償 (MIT)

Rails 向けのツール、bundler-audit, Brakeman, Codesake-dawn, RailroadMap については RailsGoat の構成ファイル (Gemfile) にパッケージを追記すれば利用可能になる。

```
1 gem 'bundler-audit'
2 gem 'brakeman'
3 gem 'codesake-dawn'
4 gem 'railroadmap'
```

Listing 4.7: Gemfile

W3AF, skipfish、ZAF は無償利用可能な Web アプリケーション用の脆弱性スキャナである。BackTrack[32] のようなペネトレーションテスト専用 OS を用いれば、簡単に利用可能である。今回は、RailroadMap から制御して開発環境上で実行するために、開発環境にこれらのツールをインストールして使用した。

```
$ sudo apt-get install libpcre3 libpcre3-dev
$ sudo apt-get install libidn11-dev
$ wget https://skipfish.googlecode.com/files/skipfish-2.10b.tgz
$ tar -zxvf skipfish-2.10b.tgz
$ cd skipfish-2.10b
$ make
```

Listing 4.8: Skipfish の導入 (Ubuntu 12.4)

W3AF

```
$ sudo apt-get install nltk SOAPpy libxml2 pysvn scapy
$ sudo apt-get install graphviz

$ apt-get install git build-essential
$ git clone https://github.com/andresriancho/w3af.git
$ apt-get install python2.7-dev python-setuptools python-pip
$ pip install PyGithub GitPython pybloomfiltermmap esmre nltk pdfminer futures scapy-real
    guess-language cluster msgpack-python python-ntlm

$ pip install -e git+git://github.com/ramen/phply.git#egg=phply

$ apt-get install graphviz python-gtksourceview2
$ pip install xdot

$ ./w3af_gui

or

$ ./w3af_console
```

Listing 4.9: W3AF の導入 (Ubuntu 12.4)

```
$ sudo update-alternatives --config java
$ wget http://zaproxy.googlecode.com/files/ZAP_2.1.0_Linux.tar.gz
$ tar -xzvf ZAP_2.1.0_Linux.tar.gz
$ sudo cp -Ra ZAP_2.1.0 /opt/zaproxy
```

Listing 4.10: ZAP の導入 (Ubuntu 12.4)

4.6.2.3 RailroadMap による外部ツールの実行管理

今回、外部テストツールの評価を実施するにあたって、RailroadMap を用いて、今回用いた外部テストツールを管理している。特に脆弱性スキャナを用いた動的テストの実行には、認証情報の付与が必要であるため、セキュリティ要求を把握している RailroadMap を用いて、実行設定ファイルの準備や、実行オプションの付与を行う。実行設定については、実行速度を優先したため、Railsgoat に最適な

[32] http://www.backtrack-linux.org/

設定になっているとは言いがたいが、安定したテストの実施が可能となる。今後、設定をチューニングすることで検知できる脆弱性は増加する可能性がある。

4.6.2.4 外部ツールの実行結果

以下に、個別の脆弱性についてその特徴と検出結果をまとめる。

A1 INJECTION Railsgoat には SQL Injection と Command Injection の脆弱性が仕込まれている。Brakeman、RailroadMap は双方の脆弱性を検出できた。

A2 BROKEN AUTH Railsgoat では独自の認証を実装しており、認証に関する3つの脆弱性が独自実装部分に仕込まれている。今回テストしたツールでは検出できなかった。

A3 XSS 文字列にサニタイズ済みのフラグを与える “html_safe” によりクロスサイトスクリプティングの脆弱性が仕組まれている。RailroadMap は “html_safe” をセキュリティ機能の無効化コマンドとして扱うため、その使用箇所に警告を発する。

A4 INSECURE DOR Direct Object Reference (DOR、直接参照) と呼ばれるの脆弱性が仕込まれている。RailroadMap は 2 つ検出したが、一つは擬陽性であった。

A5 MISCONFIG 設定ファイルに2箇所誤ったフラグ設定が仕込まれている。RailroadMap は設定の問題にはホワイトリストで対応しており、この 2 つの設定ミスを検出した。

A6 EXPOSURE 暗号化やソルトを使わないパスワードの保存と、プライバシーに関する情報の平文による保存の 2 つが仕込まれている。前者に関しては、RailroadMap は独自実装の認証機能ということで警告を発している。

A7 ACCESS Railsgoat は独自実装のアクセス制御を用いていおり、その不備。これについても、RailroadMap は独自実装の認証機能ということで警告を発している。

A8 CSRF Cross Site Request Forgery (CSRF) の脆弱性。Rails の場合は “protect_from_forgery” コマンドを適切に使用する必要がある。ApplicationController での呼び出しの有無を確認すれば良いので、静的テストでの検知は容易であり、Brakeman、RailroadMap は検出している。動的テストでも、トークンを保持している Hidden Field の確認と操作で検知が可能であり、skipfish、W3AF が検出した。

A9 COMPONENTS 脆弱性のある既知のコンポーネント（２つ）の使用。bundle-audit、codesake-dawn はすべて検出。brakeman は一つのみ検出した。

A10 REDIRECTS ユーザー入力のデータを検証なしに用いたリダイレクトの脆弱性。RailroadMap は検出。

EXTRAS Mass Assignment Mass Assignment は Rails 固有の脆弱性である。これは、静的テストで容易に検知できる。動的テストでも内部モデルの情報（ソースが公開されている場合等）があれば容易に確認可能である。ただし、一般の脆弱性スキャナでは内部のデータモデルを知らないためその検知は難しい。Brakeman が検知。

EXTRAS Constantize 外部入力からパス生成する際に、不正なファイルアクセスを許してしまう問題。対策としては入力値の検証や、フィルターを行う。検知する場合は、param[:name] のような外部入力を直接利用している箇所を検知し指摘する。コードレビューで問題がない場合は、関連する脆弱性タイプを指定し、テストケースを生成。実際の挙動に問題がないか確認する。Brakeman が検知

以上の評価結果を表 4.21 にまとめる。

表 4.21: 各種ツールによる Railsgoat のセキュリティテスト結果

Tool	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	EX
bundle-audit	–	–	–	–	–	–	–	–	2/2	–	–
Brakeman	2/2		0/N					1/1	1/2		2/2
Codesake-dawn									2/2		
skipfish								1/1			
W3AF								1/1			
ZAP											
Railroadmap	2/2		9/N	2/N	2/2	1/2	1/1	1/1	–	1/1	

4.6.3 まとめ

以上の比較から、提案手法を実装した RailroadMap が様々な脆弱性に対応可能である事がわかった。ただし、Rails 向けの静的検証ツールの性能は向上しており、検出能力も今後向上してゆくと思われる。

4.7 まとめ

以上、4 章では、提案手法を Rails 向けのツール、RailroadMap として実装することで、提案の実現可能性を検証した。また、提案手法の主要機能である、セキュリティ要求にもとづいたセキュリティ機能の静的検証 (機能 2、3)、セキュリティテストカバレッジの計測 (機能 4)、セキュリティ知識と実装との対応 (機能 5) について、様々な実アプリケーションに対して適用実験を実施した。本ツールは、MIT ライセンスのもとで Github で公開 (https://github.com/munetoh/railroadmap) しており、誰でも自由に利用することが可能である。

次の章では、この章の実験結果を元に、提案手法についてその有効性を評価する。

第5章　提案手法の評価と議論

本章では、2.6節で示した課題が、提案手法により解決が図られているかを確認する。また、関連研究、既存セキュリティテスト手法と差異についてまとめる。最後に、新たに発生した課題について整理し、その解決方法について議論する。

5.1　課題の評価

ここでは2.6節で示した4つの課題それぞれに対して、提案手法の解決方法を評価する。

5.1.1　課題1: セキュリティ要求の定義と保守

表4.10で示したように、多くのRailsで開発されたアプリケーションで、アクセス制御の実装に問題が見つかった。これらのアプリケーションの品質が、プロダクションレベルのものではないとしても、現状のRailsのアクセス制御の実装方法では、Railsの自動コード生成がアクセス制御に対応せず、自動生成コードに対して開発者による修正がアクセス制御の実装には必要となっている。その結果、開発者の実装ミスが、アクセス制御の実装での脆弱性の発生の原因となっている。また、この問題に対する効率的な対策方法がなかった。

この課題に対する提案手法の解決方法は、ツールの補助による、開発者によるセキュリティ要求の定義と保守の作業コストの低減である。開発者が暗黙的に了解しているセキュリティ要求を文書化することで、セキュリティ要求によるセキュリティ実装のチェックが可能になる。逆に、セキュリティ実装による、セキュリティ要求のチェックも可能である。これは、提案手法では、開発者定義のセキュリティ要求と、実装コードから抽出されたセキュリティ要求を並列して扱うためである。つまり、開発者がアプリケーションのセキュリティ要求を変更したい場合には、最初に実装コード上で変更を開始しても、セキュリティ要求定義の更新で変更を開始しても、最終的には双方を一致させることで、意図したセキュリティ要求の変更を実現する。

4.2.1節と、4.6.2節で、既存のサンプルアプリケーションに対して、セキュリティ要求定義を生成した。現状の実装では、MVCのModelレベルで大まかなポリシーを定義し、例外のある部分については、ControllerやViewのポリシーを

調整している。こうした調整が多く必要な部分は、主に認証部分である。例えば、Devise などの主要な外部認証モジュールを用いる場合は、ツール側で事前に作成した認証部分のポリシーを開発者に提供できる。その場合、開発者は Model レベルでのポリシー定義の設定で十分であり、このポリシー定義が開発者の大きな負担にはならない。

4.2.2 節では、セキュリティ保証が一旦完了した Web アプリケーションに変更を実施する際の、セキュリティ要求の変更と、セキュリティ実装の追加を、本提案手法を用いることで計画的に進めることができることを示した。本手法により、開発者は、初期の段階ではセキュリティ保証は考慮せずに、アプリケーションの機能変更に注力することができる。変更が完了した時点で追加的に、変更に伴うセキュリティ要求の変更を実施し、あとはツールの指示に従って（警告を修正する形で)、必要となるセキュリティ機能の配置を段階的に進めてゆけば良い。

開発者が作成したセキュリティ要求は、コードと一緒にレポジトリに保存して変更を管理することで、実装コードと一貫した形での保守が可能である。

5.1.2 課題 2: テスト駆動開発に合致した網羅的なセキュリティテスト

この課題に対する提案手法の解決方法は、テスト駆動開発に組み込んだセキュリティテストのセキュリティテストカバレッジの計測である。ブラックボックス的なセキュリティテストのアプローチで網羅的なセキュリティテストを実施すると、大量のテストケースが必要となる。しかしながら、提案手法では、アプリケーションの実装コードから、その振る舞いをモデル化するため、ホワイトボックス的なセキュリティテストのアプローチが可能である。セキュリティに関係する実装コードの場所を SINK として、SINK へのテストの有無をセキュリティテストカバレッジとして計測することで、セキュリティテストケースを最適に配置できるようになった。また、セキュリティに関係するコマンドをセキュリティ機能を実現するコマンド、SC と、その利用に脆弱性の危険があるコマンド、RC に分類する。その結果、SC については、セキュリティ要求定義と静的検証による配置の確認とを組み合わせることで、動的テストによる振る舞いの確認は、SC の機能別での実施とする事が可能となった。

4.4 節では、Rails を用いた大規模な Web アプリケーション、Foreman に対して、Foreman の既存のテストケースのセキュリティテストカバレッジの計測と、改善について実験を行った。アクセス制御実装のテストに関しては、高精度でテストカバレッジの検知が可能であり、アクセス制御テーブルを元に、セキュリティテストの配置も可能である。一方、RC の SINK カバレッジについては、Rails のテストカバレッジ計測ツールである、SimpleCov が View テンプレートのカバレッジ計測に対応しないために、限定的なものとなった。

4.3 節では、テストケースの自動生成の実験を行った。Cucumber を用いた UAT レベルのテスト記述で、アクセス制御などのセキュリティ機能のテストが自動生成可能である。ただし、Rails のテストフレームワークには、Cucumber の他にも、RSpec や、Foreman で用いられていた Rails 自体のテストフレームワークなど幾つかの選択肢がある。またテストの実装方法も、Web アプリケーションの開発毎にかなり異なるため、ツールによる自動テストケース生成を汎用的に実装する事が難しい。どの方法を用いる場合も、テストケースの記述自体は比較的簡素なため、セキュリティテストのカバレッジ計測を元に、テストケースを開発者が記述する方法が適している場合も多いと考えられる。その場合は、テスト対象となる脆弱性の理解とセキュリティテストとなる攻撃パターンの理解が重要である。これについては次節が関係する。

このように、静的なセキュリティテストと動的なセキュリティテストをモデルを介して統合していることに本手法の特徴がある。テストの基本方針は、静的テストで網羅性を確保し、動的テストで実際の挙動を確認する。また、動的なテストのためのセキュリティテスト生成には次の 2 つの目的がある。一つは実装しているセキュリティ機能が実際に働くかのテスト確認、もうひとつは静的テストで検出された問題の動的テストによる確認である。

5.1.2.1 セキュリティ機能のテスト

本手法では、網羅性の確認はセキュリティ機能の配置の問題として、静的テストで確認する（4.1.4 節）。この確認は、与えられたセキュリティ要求へのセキュリティ実装の整合性の確認である。

アプリケーションが参照する外部機能は、すべてコマンド抽象化ライブラリに登録されている。アプリケーションが参照するセキュリティ機能も、コマンドの形でコマンド抽象化ライブラリに登録されている。静的検証の過程で、アプリケーションが何処でどのコマンドを使っているのかを補足し、個々のセキュリティに関係するコマンドについてテストケースを生成する (4.1.5 節)。

現在の実装では、アクセス制御の問題を主に扱っており、表 4.4 で分類した脆弱性の検出と、そのテストケースの生成が可能である。実際のアプリケーションを用いた評価を 4.3.2 節 (Hacketyhack) と 4.4 節 (Foreman) で実施し、有効性を確認した。

5.1.2.2 セキュリティ警告の確認

提案手法及びツールでは、アクセス制御以外のセキュリティの問題にも対応している。特に、開発者によるフレームワーク提供のセキュリティ機能の無効化や、独自のセキュリティ機能の定義への対応に重点をおいた。また、本ツールを中心として、様々な既存のセキュリティテスト・ツールとの連携機能を持つ。

本提案手法のツールも含め、こうしたセキュリティテストツールには一般に擬陽性 (False-positive) の問題がある。本ツールでも、例えば完全なインフォメーションフロー解析はできておらず、疑わしい箇所は警告としてレポートしている。そうした警告が実際に問題のある脆弱性のレポートなのか、無視して良いレポートなのかの判断に動的テストを用い、実際のアプリケーションの振る舞いから陽性か擬陽性かを判断する。4.3.2 節の例では、対象アプリケーションに、脆弱性を仕込むことで、この機能の有効性について確認を行った。

提案ツール自体は既知の脆弱性のあるパッケージのチェックはサポートしないが、これについては、外部のツールと連携して、パッケージの脆弱性を把握できるようにしている（4.1.7 節）。利用しているパッケージに脆弱性が見つかった場合、本来は早急に該当パッケージを最新版に更新する必要がある。しかし、更新により機能的な問題が生じる場合などがあり、すぐには対応できないケースが発生する。こうした場合に、依存パッケージの脆弱性が実際に問題となるのか否か動的テストにより検証する機能は、実用性が非常に高い。本ツールでは未対応であるが、今後対応したい機能の一つである。

5.1.3 課題 3: セキュリティに関する情報の共有

この課題に対する提案手法の解決方法は、セキュリティ要求の定義、コマンド抽象化ライブラリによるフレームワークのセキュリティ機能の抽象化、CWE の脆弱性情報と CAPEC のアタックパターン情報と実装、との連携の 3 つの手法である。

セキュリティ要求の定義はアプリケーション開発チーム内でのセキュリティに関する情報の共有となる。一方、コマンド抽象化ライブラリの整備はフレームワークを利用する開発者全体でのセキュリティに関する情報の共有となる。

セキュリティ検証モデルの生成や、セキュリティ保証に関連するコマンドの数は表 4.14 に示すように、その数は限定的であった。Foreman ように複雑なアプリケーションの例でも表 4.13 の示すように SC,RC に分類されるコマンドの数は限定的であった。したがって、セキュリティの関するコマンドの抽出と保守は、開発者にも十分に可能であると考えられる。また、コマンド抽象化ライブラリをツールの一部として保守することで、フレームワークの標準的な機能を利用する限りは、個別のアプリケーション開発者の負担は少ないと言える。コマンドの CWE、CAPEC の分類についても、4.5 節での実験から、十分に保守可能な数であることがわかった。

5.1.4 課題 4: 変化への対応力

この課題に対する提案手法の解決方法は、課題 1 から 3 の解決策のツール化である。セキュリティ保証をアジャイルソフトウェア開発で適用する場合に、最も

重要な要素はセキュリティに関わる様々な変化への迅速な対応力である。開発者はRailroadMapを用いることで、セキュリティに関する様々な情報をメトリックスの形で利用することができる。

次に、アプリケーション開発に関連する変化を、開発アプリケーションの要求変化、フレームワークの提供する機能の変化、セキュリティ知識の変化、の3つに分類し、それぞれについて提案手法との関係をまとめる。

5.1.4.1 開発アプリケーションの要求変化

アジャイルソフトウェア開発では頻繁に要求、つまり必要とされるソフトウェアの機能が変化してゆく。Webアプリケーションであれば、新しいデータモデルや、ページ遷移が追加された場合、そうした部分へのセキュリティ要求の追加も同時に必要になる。また、機能が削除された場合には、該当するセキュリティ要求の削除も同時に必要になる。提案手法では、機能要求の変更に伴うセキュリティ要求の変更の場合は、4.2.2節で示すように、まず機能要求の変更に従って、実装作業を進める。その後、実装作業が終了した時点で、ツールによる検証を併用しながら、セキュリティ機能の実装と、セキュリティ要求の修正作業を進めていく手順を提案している。この手法のメリットは、1.2節で例として示した、実装コード上の様々な箇所に分散したセキュリティ機能の配置を、計画的かつ段階的に間違いなく進められることである。

一方、RBACによるアクセス制御を実施している場合には、新たなロールの追加や、アクセス制御リストの権限の見直しなども必要になる場合がある。この場合は、まずセキュリティ要求を変更し、その後、ツールによる検証を併用しながら、セキュリティ機能実装の修正を段階的に実施する。

どちらの場合も、ツールによる警告がなくなり、新たなセキュリティ要求のレビューが完了した時点で、更新が完了となる。

今回開発したツールは、その他のセキュリティテストツールと同様に、問題箇所を指摘するだけである。その結果を元に必要となるレビューやコードや要求の修正作業は、開発者の責任としている。

このように、本手法は、開発アプリケーションへの要求の変化と、そのセキュリティへの影響を捉え、警告とし報告することで、開発者のセキュリティ機能実装と要求定義の負担を低減している。

5.1.4.2 フレームワークの提供する機能の変化

一般に、フレームワークを用いた開発では、フレームワーク自体の変化への対応が必要となる。フレームワーク変化には、脆弱性の発覚によるセキュリティアップデート、バグ修正に伴うアップデート、機能の変更を伴うメジャーアップデート(セキュリティ機能の変更を含む)などがある。こうした変更に伴って、開発者は

様々な更新作業を実施する必要がある。これはフレームワークのみならず、アプリケーションが利用している外部パッケージについても当てはまる。

提案手法では、アプリケーションが依存するフレームワークや外部パッケージが提供する機能をコマンドライブラリの形でまとめている。したがって、フレームワークや外部パッケージ側で生じた変更は、コマンドライブラリの修正で対応する必要がある。

5.1.4.3 セキュリティ知識の変化 (外部環境の変化)

これは、開発するアプリケーションに依存するが、例えば法令や規制の変更などがアプリケーションの要求の変更につながるケースが想定できる。現在提案しているフレームワークでは、そうした前提条件と要求とを取り扱う仕組みは提供できていない。例えば、HIPPA や PCI-DSS への順守の確認ができるようにフレームワークを拡張することも、今後検討する必要がある。これは、ある産業ドメインのセキュリティ知識を扱えるようにすることを意味する。

もう一つは、新たな攻撃手法への対応である。これは前節とも関係するが、コマンドライブラリの更新と、本ツール側での対応で対処可能であると考える。

5.1.4.4 ツール自体の保守

本ツールも Ruby のアプリケーションの一つであり、バグの修正や、新機能の追加が絶えず必要となる。実際に、様々な Rails アプリケーションについて本ツールの適用を試みたが、新たな実装スタイルへの対応が必要となり、コードの修正や機能追加が必要となった。Ruby は様々な記述が可能である。また新規の機能パッケージが利用されている場合には、そのパッケージに対応できるように、本ツールの修正が必要となる場合が多く発生した。Ruby の構文については、未対応の構文があった場合にはエラーとして報告している。コードからのセキュリティ検証モデル生成についても、遷移先が見つからない場合はエラーを報告する。

そうした個別の問題を細部にわたってサポートするか、開発者の判断で補正するかの判断は難しいが、基本的には、本ツールの実装が出来るだけ単純になる形で個別の問題への対応を進めてゆくことにした。本ツール自体もアジャイルソフトウェア開発に対応し、テスト駆動で機能拡張を進めている。

5.2 比較

5.2.1 関連研究との比較

様々な既存研究が、本研究と関連する。その中で、Web アプリケーションのモデリングとモデル駆動開発、セキュリティオントロジー、Web アプリケーションのセキュリティテストの 3 つについて既存研究との差異についてまとめる。

5.2.1.1 Web アプリケーションのモデリングとモデル駆動開発

Web アプリケーションのモデリングに関しては、Andrew らが Web アプリケーションのソースコードから状態遷移モデルを生成してテストを実施する手法を提案している [12][13]。この提案では、粒度の小さい状態で Web アプリケーションを表現し、詳細なテストを実施するため、Web アプリケーションの規模が大きくなると、モデルもそれに比して大きくなる。これに対して、提案手法の制御フローモデルの粒度は MVC を基本として、状態の粒度は Web アプリケーションのページ（URL）レベルである。また、セキュリティ保証に必要な機能だけをモデル化する。

Sprenkle らは、アプリケーションの実行とレースからモデルを生成する手法を提案している [58]。この手法では、アプリケーションのソースコードのパースは不要であるが、モデル生成は、アプリケーションを動作させるテストケースに依存する。提案手法は、コマンド抽象化ライブラリを用いることで、ソースコードの検査スコープを MVC を実現するコードのみに絞ることで、アプリケーションの振る舞いのモデル化を実現している。一方、動的テストに関しては、テストのコードカバレッジを、セキュリティテストのカバレッジの計測に利用している。

5.2.1.2 セキュリティオントロジーの活用に関する研究

Wang らは、ソフトウェア製品のセキュリティ特性を、関連する CVE、CWE、CPE、CAPEC の情報から分析する手法を提案している [65]。この手法は、単純に利用しているパッケージの過去の脆弱性情報からセキュリティの特性を得る。この情報の主な利用者は IT システムの管理者や調達者である。提案手法でも、CWE などを利用する。理由はソフトウェアセキュリティのオントロジーとしてツールが利用しやすいためと情報が公的な組織により保守されていることである。違いは、実装コードで用いるコマンドレベルで対応付けを行うために、開発者が直接参照できることである。過去に CVE に登録された脆弱性情報を参照することで、正確な対応付けが可能であるが、そうした前例がなくても、一般論としてコマンドとオントロジーを対応付ける事が可能である。したがって、過去の事例ではなく、現在の実装をベースにそのセキュリティ特性を得ることができる。

Vanciu らは、設計から実装レベルの様々な脆弱性に対して、アーキテクチャレベルとコードレベルのアプローチでのテストケースの有効性を比較している [63]。その結果、アーキテクチャレベルでの表現は、コードを読むよりもシステムの理解が容易であるにもかかわらず、多くのツールがコードレベルに注力していると指摘している。提案手法もコードを起点としてセキュリティ的な解析を実施するが、一旦セキュリティ検証モデルの形で対象の振る舞いを抽象化することで、セキュリティ上の特性の解析を容易にしている。

5.2.1.3 Web アプリケーションのセキュリティテストに関する研究

Rails を対象にしたセキュリティテストの研究では、静的検証による脆弱性発見の手法の提案が主である。一般に、PHP や Ruby、Python のような動的スクリプト言語を用いる Web アプリケーションの静的検証は難しい。

Sun らは PHP の Web アプリケーションのソースコードのアクセス制御実装から Sitemap を作成し、アクセス制御の脆弱性を検証する手法を提案している [59]。これに対して、提案手法では、開発者が定義するセキュリティ要求 (アクセス制御リスト) を基準とし、実装コードとの一貫性を確認するアプローチを取る。

Chaudhuri らは、Rails の脆弱性をソースコード解析とシンボリック実行で発見する手法を提案している [23]。パーサーには OCaml を用いているため、開発者は別途ツールを実行する環境を準備しなければならない。これに対して、RailroadMap は、開発環境との整合性を重視して、すべての実装を Ruby 言語で行った。

Doup らは、execution after redirect (EAR、CWE-698) 脆弱性を静的検証で発見する手法を提案している [28]。ただ、EAR の脆弱性は、コードスコープ的には非常に局所的であり、検知は容易である。

Near らはツール、Derailer を開発している [52]。Derailer は Ruby のランタイムをハイジャックしシンボリック実行することで、開発者と対話的に、アプリケーションデータに対するセキュリティポリシーを定義する。提案手法も、セキュリティ要求定義を開発者が実行する点は、Derailer と同じである。ただし、アプリケーションの振る舞いは、コマンド抽象化ライブラリを用いた、セキュリティ検証モデルの生成と、モデル上でのポリシー定義の形で実施している。

5.2.2 既存のセキュリティテスト手法の比較

RailroadMap の開発開始当初 (2011 年) は、Rails 向けのセキュリティツールは存在しなかった。その後、Rails 向けの静的検証ツールとして、Brakeman や Codesake-dawn 等のツールが公開されでいる。これらは、コマンドラインツールとして簡単に実行可能であり、特に Brakeman は OWASP でもその利用が推奨されてる (2015 年時点)。こうしたツールを用いることで、Rails アプリケーション

の静的解析から、XSS のような Web アプリケーション固有の問題が検出できる。また、利用コンポーネントに CVE に登録されている既知の脆弱性が存在しないか、そのバージョンをチェックする。

Railroadmap も静的検証を用いるが、セキュリティ視点でのアプリケーションの解析とセキュリティテストカバレッジの計測の自動化を目指している。4.6 節で示したように RailroadMap も Brakeman のような静的検証機能を持つが、主な目的は、プログラマのセキュリティエンジニアリング活動のサポートである。アプリケーションで、どの様なセキュリティ機能が用いられ、それがどの様なセキュリティ要求を元に選択されているかを、ソースコードから抽出し、要求定義との一貫性を確認する。反復開発の中で、セキュリティ要求定義と実装の間に不一致が発生した場合にそれを自動検知し、デザインと実装に起因するセキュリティ問題双方に単一のツールで対応する。

セキュリティ保証のベスト・プラクティスは、様々なセキュリティ保証手法を組み合わせて開発に積極的に組み込んでゆくことだと考える。実際に、セキュリティテスト・ツールには、検出する脆弱性に制限があり、複数のツールを相補的に活用することでセキュリティ保証のレベルを高めることができる。提案手法の実装で作成した RailroadMap はセキュリティ要求のレベルから対象アプリケーションを把握するため、包括的なセキュリティ保証フレームワークを、Rails を用いた Web アプリケーション開発に提供することが可能である。表 5.1 に開発プロセスと、4 つのセキュリティ保証手法との関係を整理した。

表 5.1: セキュリティテスト機能比較

開発プロセス	マニュアル レビュー/テスト	Brakeman Codesake	W3AF skipfish	RailroadMap
セキュリティ 要求定義	可能	既知の脆弱性	既知の脆弱性	コードから 要求を生成、検証
静的テスト (設計)	対応 [1]	未対応	–	対応
静的テスト (実装)	対応 [1]	対応		対応/連携 [3]
動的テスト (設計)	対応 [2]	未対応	未対応	対応
動的テスト (実装)	対応 [2]	対応	対応	対応/連携 [3]

1) コードレビュー

2) ペネトレーションテスト

3) その他のツールと連携可能

5.2.3 BSIMM との比較

2.3.4 節で BSIMM で成熟度評価に用いられているセキュリティ保証プロセスについて、アジャイルソフトウェア開発との整合性を分類した。ここでは本提案手

法との関係を議論する。

GOVERNANCE: STRATEGY AND METRICS 基本的には、開発手法に無関係であるが SM2.2 については、RailroadMap の警告をメトリックスと活用することで、アジャイルソフトウェア開発との整合性が実現できる。

GOVERNANCE: COMPLIANCE AND POLICY 基本的には、開発手法に無関係である。

GOVERNANCE: TRAINING 基本的には、開発手法に無関係である。

INTELLIGENCE: ATTACK MODELS セキュリティ要求を文書化するため、AM1.2 の整合性が高まる。また、攻撃のテストケース記述により、AM1.3、AM1.4、AM2.1 の整合性が高まる。開発者全体で共通のセキュリティツールを用いることで、最新のセキュリティ問題への対応が早まれば、AM1.5、AM3.1、AM3.2 の整合性が高まる。

INTELLIGENCE: SECURITY FEATURES AND DESIGN セキュリティ要求を文書化するため、SFD1.1、SFD3.3 の整合性が高まる。最新の Web アプリケーションフレームワークを用いると SFD2.1 の整合性が高まる。

INTELLIGENCE: STANDARDS AND REQUIREMENTS 基本的には、開発手法に無関係である。

SSDL TOUCHPOINTS: ARCHITECTURE ANALYSIS セキュリティ要求を文書化するため、AA1.1、AA1.2 の整合性が高まる。セキュリティ検証モデルにより AA.2.2 の整合性が高まる。セキュリティツールにノウハウが集約される事は、AA3.1 の整合性を高めることに近い。

SSDL TOUCHPOINTS: CODE REVIEW セキュリティ検証モデルにより CR1.4 の整合性が高まる。

SSDL TOUCHPOINTS: SECURITY TESTING) 提案手法により ST1.3 の整合性が高まる。外部テスト連携は ST2.1 の整合性を高める。ダッシュボードにより ST2.4 の整合性を高める。

DEPLOYMENT: PENETRATION TESTING 動的テスト生成機能は PT1.3 の整合性を高める。

DEPLOYMENT: SOFTWARE ENVIRONMENT 基本的には、開発手法に無関係である。

DEPLOYMENT: CONFIGURATION MANAGEMENT AND VULNERABILITY MANAGEMENT 基本的には、開発手法に無関係である。

BSIMM は単一組織のセキュリティの取り組みの成熟度を測る指標であるが、Railsや本提案ツールのようなオープンソースによるソフトウェア開発では、BSIMMの組織の枠はオープンソース・コミュニティとして捉えるべきであろう。つまり、組織横断の情報共有は、コミュニティにおける情報共有と言える。その中で、提案手法のようなセキュリティテストツールの機能を絶えず更新し充実させることは、そのコミュニティのセキュリティ保証に関する成熟度を高める事になる。

5.2.4 各種のトレードオフについて

提案手法の実現及び実装にあたり、様々なトレードオフがあった。この節ではトレードオフの種類と今回選択した対応についてまとめる。

コマンド抽象化ライブラリの作成と精度： 提案手法ではコマンド抽象化ライブラリの作成が本手法を利用する上での前提となる。未知のコマンドについてはツール実行時に検出し、ライブラリへの追加のためのテンプレートを表示する。そのため、コマンド抽象化ライブラリへの登録自体の手間は少ない。その際に、そのコマンドの持つセキュリティ属性、モデル生成で必要となる属性については、開発者が定義を追加する必要がある。アプリケーション側で作成して利用しているコマンドについては、開発者がその機能について把握しているはずであり、開発者自身でコマンドに対して正しい分類を定義する必要がある。一方、フレームワーク側のコマンドについては共有のライブラリとして、ツール側で整備し、利用者間で共有する仕組みとした。セキュリティに関しては、当初はコマンドを独自の分類で扱っていたが、この分類作業がライブラリ作成のうえで、大きな負担であった。その後、CWE,CAPECの定義と連携することで、定義の作成が不要となり、割当も番号の指定のみで良いため作業が簡略化できた。

コマンド抽象化ライブラリで正しく定義できていない場合、ツールは正しい解析ができない。定義の正しさを確認する一つの方法は、過去の事例（CVEやレポジトリで問題がコードレベルで確認できる事例）を用いた評価である。これについては、過去の事例をテストケースとして取り込みながら、ツールの開発を行うことで対応した。また、4.5 節で示した実装コードと CWE、CAPEC との対応の確認は、ツールが想定している脆弱性が取り扱えているかを確認する意味で有効であった。

セキュリティ検証モデルの精度、制御フロー： Rails の場合、実装コードと MVC の各状態との対応が一意に対応するため、制御フローの各状態は正しく抽出することができる。遷移については、遷移に関係するコマンドがコマンド抽象化ライブラリで定義されていれば抽出可能である。

セキュリティ検証モデルの精度、データフロー： Ruby のコードの静的解析には限界があるため、Rails の場合、データフローの完全な解析は難しい。現在の実装では、コード上で解析可能範囲でデータフローモデルを構築する。

セキュリティ上問題となるのは、RC 近傍の変数の扱いである。この際に SC との対応が不明なデータパスについては警告となる。

セキュリティ検証モデルの精度、遷移条件： 制御フローモデルで遷移が正しく抽出出来た場合、View から Controller への遷移については、ナビゲーションエラーの検出精度が上がる。アクセス制御については、任意の状態からの Controller への遷移が問題となるため、モデル上での Controller への遷移パスや遷移条件は関係ない。

例えば、セキュリティ検証モデルを B メソッドに変換し、別のツールを使ってアプリケーションの振る舞いの評価を行う場合には、正しいアプリケーションの振る舞いを模すためにできるだけ正確な内部変数の定義と遷移条件の指定が必要となる。

Rails の場合、コマンドの分類と MVC コードの静的検証でセキュリティ検証モデルを構築することが可能である。正しくモデル化できない一部の箇所については、ツールの精度を求めるより、開発者が補足できる仕組みを用意した。一般に、ツールの精度を高めることと、ツールの実行速度にはトレードオフとなるが、ここでは、ツールを出来るだけ軽量に実装することを優先した。

ツールの形態： 提案手法をツール化するにあたり、当初は単体のツール（実行コマンド）での機能の実現を目指した。これは静的解析ツールと同様の使用方法を想定したためである。典型的な Rails のアプリケーションの実装形態については、単体ツールで対応可能である。

しかしながら、大規模な Rails アプリケーションでは、アプリケーション独自の実装方式（特にコードの配置方法）を用いる場合があり、ツールの機能を個別のケースにも対応できるように拡張することは難しい。こうしたケースに本手法を適用する一つの方法は、個別のセキュリティ検査機能をライブラリ化し、アプリケーション開発側で利用することで、独自のセキュリティ保証を実行する環境を構築することである。そのためには、現在のツール内部の処理をモジュール化し、内部のデータ構造を整理する必要がある。

以上のようなトレードオフは、対象とするアプリケーション（及びアプリケーションフレームワーク）で異なる可能性がある。

5.3 今後の課題

この節では、提案手法で十分に解決できなかった問題についてまとめる。ここでは、ツールの技術的な課題、開発作業としての課題、アプリケーションの実装方法に関する 3 つの課題の今後のアプローチについて考察する。

5.3.1 ツールの技術的課題

RailroadMap の実装と実験を通して、セキュリティ検証モデル生成の自動生成の精度に課題が残った。Ruby の記述には自由度が多く、セキュリティ検証モデルを、すべて自動には生成できていない。そのため、制御フローモデル上で遷移先が不明な遷移は、開発者が手動で補正する仕組みをサポートしている。Rails のサンプルコードに近い、典型的な実装の場合には、こうした補正箇所はかなり少ないが、4.4 節の実験で用いた Foreman のように、大量の独自機能を実装しているアプリケーションの場合、汎用的なツールで対応することは難しい。

この問題に対する一つの解決方法は、単純なコマンドライン上での実行ツールではなく、開発者が自身のアプリケーションのセキュリティ保証に適した形で、カスタマイズできる仕組みが有効であると考えられる。そのための一つの方法は、ツール内部の機能のモジュール化と、個別のアプリケーションの開発に合致したスクリプトを利用して、開発者が独自のセキュリティ保証環境を構築することである。この場合、アプリケーション実装コードの解析や、個別のセキュリティ保証作業を、ある粒度で整備する必要がある。

5.3.2 開発作業としての課題

開発作業としての課題には、少なくとも次の 2 つの課題が見つかった。一つは、実装コードレベルでの、セキュリティ問題の確認及び修正作業の容易さの確保。もう一つは、セキュリティ検証モデルの可視化である。

提案手法ではメトリックスを用いて、セキュリティ保証の作業を計画的に実施する方法を示した。4.4 節で示したように、提案手法ではマクロな各種メトリックスを提示することで、段階的にセキュリティ保証作業を進めてゆく事ができる。しかしながら、ツールがレポートする個別の問題の解決には、その問題とアプリケーション実装とについての深い知識が必要である。開発者に、問題と関連した CWE や CAPEC の情報を提示することは可能となったが、CWE や CAPEC での記述は Rails に特化したものではない。そのため、開発者は問題を理解して、実装コードやテストの修正を実施するには、依然として開発者の能力に依存せざるを得ない。この問題は、前節で挙げた課題とも関連する。つまり一回のツール実行で検査する対象がアプリケーション実装全体であるため、大量に問題が存在する場合に、大量

の警告がレポートされてしまい、個別の問題の修正が難しくなる。個別の問題に対応するためには、逆にミクロな視点で、問題箇所の周辺の実装を理解して、コードの修正を正しく行う必要がある。

こうした、特定の問題を深掘りする機能として、3.3.3 節と 4.1.5 節でトレース機能について簡単に説明した。提案手法では、コマンド実行により対象のアプリケーション全体を一度に処理している。そして、修正対象のエラーや警告について、その関連情報を詳しくダンプすることで、問題の詳細な理解を行うのがトレース機能である。

こうした課題に対応するのが、個別の問題を柔軟に取り扱える、セキュリティテスト手法である。一つの方法は、前節でも指摘したように、セキュリティ保証の機能を API として整備し、テストスクリプトで柔軟に問題を扱える仕組みを実現することである。つまり、セキュリティ保証の作業を適切な粒度で API 化（手続き化）し、ライブラリとして提供することで、アプリケーションテスト環境にセキュリティ保証に組み込む。

次に、セキュリティ検証モデルの可視化についての課題と、今後の研究方針について整理する。アプリケーションが大規模になると、制御フローモデルやデータフローモデルも巨大になる。その結果、生成されたモデルを開発者は簡単には確認できなくなる。現在は表の形で HTML のレポートを生成しているが、HTML5 を用いた検索機能を追加しているため、個別の遷移やデータの流れを表で確認することは容易である。しかしながら、モデルが示す動的な振る舞いを、表形式で確認することは難しい。

一方、セキュリティ検証モデルの制御フローやデータフローを「状態遷移図 (State Diagram)」の形で出力することは可能である。例えば、Graphiviz の DOT などを用いて作図が可能であるが、Web アプリケーションの制御フローには多くのループが存在するため、複雑に絡まった図が生成される事になる。これは、アプリケーション全体の規模の確認には十分に使えるが、個別の問題解決のための検討を、グラフ上で行うことは難しい。作図についても、人間が目でレビューをすることを前提として、どのような表示方法が良いか十分に検討する必要がある。

ソフトウェアの関心部分のみを抜き出す手法として「プログラムスライシング」や「モデルスライシング」が知られている [14]。セキュリティの問題を個別に解析する際にも、こうした手法を用いることで開発者に問題箇所をわかりやすく提示できる可能性がある。

5.3.3 アプリケーションの実装方法に関する課題

4.5.3 節や、4.6 節で扱ったアプリケーション、Railsgoat に仕組まれた脆弱性の多くは、コマンドの実装に関する物であった。アプリケーション独自のでセキュリティ機能を実装する場合には、開発者がセキュリティコマンド自体の実装が安

全であることを検証する必要がある。一方、セキュリティ機能として、フレームワークや、広く利用されているパッケージを利用する場合には、開発者は利用するパッケージに脆弱性が無いことに留意すればよい。また、本手法を適用する場合には、利用しているパッケージに対応するコマンド抽象化ライブラリの定義が正しいことが重要である。

セキュリティに関するコマンドを実装する場合に、コマンドが提供するセキュリティ機能の粒度について配慮することで、本手法で扱いやすいコマンドとなる。例えば、一つのコマンドが、ある特定のセキュリティ機能に対応していると、ツールによるセキュリティ機能の特定や、テストカバレッジの計測に都合が良い。これはテストケース記述にも該当し、セキュリティテストの記述に適した、セキュリティテスト用のコマンド定義が存在する。つまり、実装コードやテストケース上でセキュリティの問題をトレースしやすいように、コマンドを作成する。 これらを、より定式化し、セキュリティ機能を分類することで、コマンドレベルの抽象化によるセキュリティ保証に適した、実装のガイドラインを定義できる可能性がある。

5.4 まとめ

以上、5 章では、提案手法が 2.6 節で挙げた 4 つの課題を解決できたか評価するとともに、関連研究や既存手法と提案手法の比較をまとめ、今後の課題を 4 つ指摘した。提案手法は、最初に掲げた課題を十分に解決する、新しいセキュリティ保証の仕組みを実現しているといえる。

第6章　結論

6.1　本研究の成果

最後に本研究の貢献についてまとめる。

本研究では、Web アプリケーションのセキュリティ保証を包括的に実現する手法として、コマンド抽象化ライブラリを用いた、新しいセキュリティ保証手法を提案した。具体的には、セキュリティ機能やアプリケーションの振る舞いに関する情報を抽象化し、コマンド抽象化ライブラリの形で整備することで、Web アプリケーションの実装コードから、Web アプリケーションのセキュリティ問題を扱うモデル定義 (セキュリティ検証モデル) を効率的に生成できること、セキュリティ知識の集積と、セキュリティオントロジーとの連携ができること、セキュリティコマンドを SC と RC に分類し、その SINK カバレッジを用いることで、設計、実装双方の問題に関連するセキュリティテストのテストカバレッジ計測が可能となることを、ツールの実装と実験を通して確認した。

アジャイルソフトウェア開発で課題であった、セキュリティ要求の定義と保守には、セキュリティ検証モデルを介した、セキュリティ要求とセキュリティ実装の双方の一貫性を確認によって対応した。

セキュリティテストのテスト駆動開発への組み込みについては、対象の実装コードの静的な検証と、動的テストとの組み合わせで実現する。静的検証ではセキュリティ要求と実装との一貫性を確認するとともに、SC,RC を元にソースコード上の SINK を特定し、動的テストの結果と連携することで、SINK カバレッジを導出し、セキュリティテストのカバレッジとすることで、開発者が作成したセキュリティテストの網羅性を確認できる仕組みを実現した。これにより、SC,RC それぞれの SINK に対して、より少ないテストケース数でセキュリティテストの実施が可能となる。特に RC により、実装コード中のセキュリティ上の危険箇所を把握することで、実装に起因するセキュリティ問題にも対応可能である。SC については、その配置を静的検証で網羅的に確認し、個別のセキュリティ機能の単体の振る舞い、アプリケーションとしての実際の振る舞い双方について動的テストで確認する。このように、TDD の受け入れテストフレームワークを利用してセキュリティテストを実施することで、開発者が直接セキュリティ保証を実施できるようになる。

フレームワークや利用パッケージが提供するセキュリティ機能は、コマンド抽象

化ライブラリの形で整理されることで、そのアプリケーション・フレームワークの開発者全体で共有することができるようになる。また、より一般的な（ソフトウェアの脆弱性に関する）セキュリティ知識については、コマンドと CWE,CAPEC の情報をリンクすることで、実装コードの視点から、セキュリティのオントロジーを確認することができるようになった。

最後に、アジャイル開発で重要な変化への対応についてまとめる。提案手法では、要求の変化とコード実装変化の一貫性をツールによりチェックする仕組みを提供することで、要求側がトリガーとなる変更も、コード実装側がトリガとなる変更も同様に扱う。セキュリティテストについても同様で、カバレッジを元に、実装コードとテストケースの一貫性をチェックする。つまり、反復開発の中で、要求、実装、テストのそれぞれのセキュリティに関する変更を捉え、最終的に、要求、実装、テストで整合させることで、セキュリティ保証を実現する。これは、要求、実装、テストを単一のセキュリティ保証ツールで取り扱うことで、初めて実現できる。

6.2 今後の研究課題と方向性

本研究の今後の方向性は大きく 2 つあると考える。ひとつは、より汎用的な手法として成熟させてゆく方向である。本研究では、Web アプリケーションフレームワーク、Ruby on Rails を実例として開発と評価を実施したが、その他の Web アプリケーションフレームワークや Web 以外のアプリケーション開発への適用である。そのためには、プログラミング言語への依存性を出来るだけ少なくして、提案手法の汎用性を高める必要がある。

もうひとつの方向性は、セキュリティ保証の対象をアプリケーション開発から、より複雑で大規模なシステムやインフラレベルへの拡張である。例えば、Web アプリケーションを、稼働するフレームワークや OS、インフラも含めた、全体のシステムレベルでセキュリティ保証を行うために、より包括的なセキュリティ保証ケースの構築を進めることで、個別のアプリケーションが持つべきセキュリティ要求をより明確に定義、検証できるようになる。現在のアジャイルソフトウェア開発ではセキュリティ要求は開発者（チーム）が正しく定義する必要があるが、アプリケーションドメインに関する様々な標準や規制を元に、的確なセキュリティ要求を定義する事ができれば、アプリケーション開発者の負担の低減と、開発されるアプリケーションのセキュリティ向上につながるものと思われる。それには、様々なレベルのセキュリティ知識をシームレスに連携する仕組みが必要となる。

謝辞

本研究を進める機会とご指導をいただきました国立情報学研究所の吉岡信和准教授に心から感謝いたします。

またお忙しい中、本博士論文の審査委員をご快諾くださり、ご指導をいただきました、国立情報学研究所の中島震教授、胡振江教授、石川冬樹准教授、早稲田大学の鷲崎弘宜准教授に深く感謝いたします。

研究を進める過程でも多くの方にお世話になりました。特に情報セキュリティ大学院大学の大久保隆夫准教授、神奈川大学の海谷治彦教授には研究内容についてご助言いただき深く感謝いたします。

大学院に進学するにあたっては、サポートしていただいた日本アイ・ビー・エムの上条昇さん、片山泰尚さん、細川浩二さん、工藤道治さんに感謝いたします。

最後に、大学院生としての生活を支援してくれた家族にこの場を借りて感謝の意を表します。

略語

BDD Behavior-driven development

BSIMM The Building Security In Maturity Model

CC Common Criteria (ISO/IEC15408)

CGI Common Gateway Interface

CoC Convention over Configuration

CSRF Cross site request forgeries

CAPEC Common Attack Pattern Enumeration and Classification

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

DRY Don’t Repeat Yourself

DSL Domain specific language

EJB Enterprise JavaBeans

ERB Embedded Ruby (eRuby)

FISMA Federal Information Security Modernization Act

MAST Model-assisted security testing

MBST Model-based security testing

MDD Model-driven development

MVC Model-View-Contoller

NIST National Institute of Standards and Technology (US)

OCL Object Constraint Language

SAMM Software Assurance Maturity Model

SC Security Command

PCI-DSS Payment Card Industry Data Security Standard

PEP Policy Enforcement Point

PDP Policy Decision Point

PHP Hypertext Preprocessor

PP Protection Profile (ISO/IEC15408)

RBAC Role-based access control

RC Risky Command

RDBMS Relational database management system

SCAP Security Content Automation Protocol

SDLC Software development life cycle

ST Security Target (ISO/IEC15408)

TDD Test-driven development

UAT User acceptance testing

URL Uniform Resource Locator

XP eXtreme Programming

XSS Cross-site scripting

用語

Attack surface ソフトウェアやシステムへの攻撃箇所

Dataflow model Web アプリケーションのデータフローを表現するモデル

Command Abstraction Library (CALib) コマンド抽象化ライブラリ

Control flow model Web アプリケーションの挙動を表現する状態遷移モデル

Navigation model Control flow model と同じ (古い Version の RailroadMap で使用)

Vulnerability 脆弱性、セキュリティ上問題となるソフトウェアの欠陥 (バグ)

発表文献

Refereed papers published in journals or books (first author)

- Seiji Munetoh and Nobukazu Yoshioka. Method using command abstraction library for iterative testing security of web applications.Int. J. Secur. Softw. Eng., 6(3) 26–49, July 2015.

Refereed papers published in journals or books (co-author)

- 吉岡 信和, 大久保 隆夫, 宗藤 誠治, セキュリティソフトウェア工学の研究動向, コンピュータソフトウェア Vol.28,No.3 pp.43-60, (2011)

Refereed papers published in international conference proceedings (first author)

- Seiji Munetoh and Nobukazu Yoshioka, RAILROADMAP: An Agile Security Testing Framework for Web-application Development, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), March 2013, pp. 491 - 492 (poster)
- Seiji Munetoh and Nobukazu Yoshioka, Model-Assisted Access Control Implementation for Code-centric Ruby-on-Rails Web Application Development, 2013 Eighth International Conference on Availability, Reliability and Security (ARES), 6 Sept. 2013, pp.350-359

参考文献

[1] Extreme programming: A gentle introduction.

[2] OWASP Web Application Security Requirements 2.0. Standard, OWASP.

[3] What is scrum?, 2010.

[4] Payment Card Industry Data Security Standards version 3.0. Standard, PCI Security Standards Council, Dec. 2013.

[5] FIPS 200. Minimum security requirements for federal information and information systems. *RubyConf 2011*, March 2006.

[6] Walid Al-Ahmad. Building secure software using xp. *Int. J. Secur. Softw. Eng.*, 2(3):63–76, July 2011.

[7] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. A survey of analysis models and methods in website verification and testing. In *Proceedings of the 7th International Conference on Web Engineering*, ICWE'07, pages 306–311, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automated verification of role-based access control security models recovered from dynamic web applications. In *Proceedings of the 2012 IEEE 14th International Symposium on Web Systems Evolution (WSE)*, WSE '12, pages 1–10, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Recovering role-based access control security models from dynamic web applications. In *Proceedings of the 12th International Conference on Web Engineering*, ICWE'12, pages 121–136, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 100–109, New York, NY, USA, 2012. ACM.

[11] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 662–671, Piscataway, NJ, USA, 2013. IEEE Press.

[12] A. Anneliese Andrews, Jeff Offutt, and T. Roger Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345, 2005.

[13] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using fsmweb to test web applications. *Inf. Softw. Technol.*, 52(1):52–66, January 2010.

[14] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based model slicing: A survey. *ACM Comput. Surv.*, 45(4):53:1–53:36, August 2013.

[15] Dejan Baca and Bengt Carlsson. Agile development with security engineering activities. In *Proceedings of the 2011 International Conference on Software and Systems Process*, ICSSP '11, pages 149–158, New York, NY, USA, 2011. ACM.

[16] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 332–345, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[18] Kent Beck. Manifesto for agile software development, 2004.

[19] Konstantin Beznosov and Philippe Kruchten. Towards agile security assurance. In *Proceedings of the 2004 Workshop on New Security Paradigms*, NSPW '04, pages 47–54, New York, NY, USA, 2004. ACM.

[20] Barry Boehm. Revisiting software engineering economics. *EQUITY 2007 kenote*, March 2007.

[21] Gustav Boström, Jaana Wäyrynen, Marine Bodén, Konstantin Beznosov, and Philippe Kruchten. Extending xp practices to support security

requirements engineering. In *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, SESS '06, pages 11–18, New York, NY, USA, 2006. ACM.

[22] Julien Botella, Fabrice Bouquet, Jean-FranÃğois Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-based testing for cryptographic components - lessons learned from experience. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 192–201, 2013.

[23] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 585–594, New York, NY, USA, 2010. ACM.

[24] Justin Collins. Keeping rails applicationson track with brakeman. *RailsConf 2012*, April 2012.

[25] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 267–282, Berkeley, CA, USA, 2009. USENIX Association.

[26] Thanh Binh Dao and Etsuya Shibayama. Coverage criteria for automatic security testing of web applications. In *Proceedings of the 6th International Conference on Information Systems Security*, ICISS'10, pages 111–124, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] Thanh Binh Dao and Etsuya Shibayama. Security sensitive data flow coverage criterion for automatic security testing of web applications. In *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, pages 101–113, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. Fear the ear: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 251–262, New York, NY, USA, 2011. ACM.

[29] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[30] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[31] Michael Joseph Edger. Laser: Static analysis for ruby, in ruby. *RubyConf 2011*, 2011.

[32] Michael Joseph Edger. Static analysis for ruby in the presence of gradual typing. *Dartmouth Computer Science Technical Report TR2011-686*, 2011.

[33] Gencer Erdogan, Per Håkon Meland, and Derek Mathieson. *Agile Processes in Software Engineering and Extreme Programming: 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010. Proceedings*, chapter Security Testing in Agile Web Application Development - A Case Study Using the EAST Methodology, pages 14–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[34] Matthew Finifter and David Wagner. Exploring the relationship betweenweb application development tools and security. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[35] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, pages 27–32, New York, NY, USA, 2008. ACM.

[36] FranÃğois Gauthier and Ettore Merlo. Fast detection of access control vulnerabilities in php applications. *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 447–256, 2012.

[37] Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 235–244, New York, NY, USA, 2010. ACM.

[38] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[39] Systems and software engineering – Software life cycle processes. Standard, International Organization for Standardization, Geneva, CH, 2008.

[40] The Common Criteria for Information Technology Security Evaluation. Standard, International Organization for Standardization, Geneva, CH.

[41] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[42] Jan Juerjens. *Secure Systems Development with UML*. SpringerVerlag, 2003.

[43] Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 322–331, New York, NY, USA, 2005. ACM.

[44] Jan Jürjens. Model-based security testing using umlsec. *Electron. Notes Theor. Comput. Sci.*, 220(1):93–104, December 2008.

[45] Hossein Keramati and Seyed-Hassan Mirian-Hosseinabadi. Integrating software development security activities with agile methodologies. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 749–754, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Vidar Kongsli. Towards agile security in web applications. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 805–808, New York, NY, USA, 2006. ACM.

[47] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. *The Fourth International Workshop on Security Testing, SECTEST2013*, pages 445–452, 2013.

[48] Dominic Letarte and Ettore Merlo. Extraction of inter-procedural simple role privilege models from php code. *2009 16th Working Conference on Reverse Engineering*, pages 187–191, 2009.

[49] Pratyusa K. Manadhata and Jeannette M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering (Volume:37, Issue: 3)*, 2011.

[50] Gary McGraw. Security fatigue? shift your paradigm. *IEEE Computer, (Volume:47, Issue:3)*, pages 81–83, 2014.

[51] J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. Improving web application security: Threats and countermeasures roadmap. *Microsoft Corporation*, June 2003.

[52] Joseph P. Near and Daniel Jackson. Derailer: Interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 587–598, New York, NY, USA, 2014. ACM.

[53] Torstein Nicolaysen, Richard Sassoon, Maria B. Line, and Martin Gilje Jaatun. Agile software development: The straight and narrow path to secure software? *Int. J. Secur. Softw. Eng.*, 1(3):71–85, July 2010.

[54] Security Considerations in the System Development Life Cycle. Standard, National Institute of Standards and Technology, NIST, 2008.

[55] Ben Poweski and David Raphael. *Security on Rails (The Pragmatic Programmers)*. Pragmatic Bookshelf, 2008.

[56] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. An empirical analysis of input validation mechanisms in web

applications and languages. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1419–1426, New York, NY, USA, 2012. ACM.

[57] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

[58] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 230–239, Washington, DC, USA, 2011. IEEE Computer Society.

[59] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[60] Larry Suto. Analyzing the Effectiveness and Coverage of Web Application Security Scanners. Technical report, Oct. 2007.

[61] Larry Suto. Analyzing the Accuracy and Time Costs of Web Application Security Scanners. Technical report, Feb. 2010.

[62] A. Tappenden, P. Beatty, and J. Miller. Agile security testing of web-based systems via httpunit. In *Proceedings of the Agile Development Conference*, ADC '05, pages 29–38, Washington, DC, USA, 2005. IEEE Computer Society.

[63] Radu Vanciu, Ebrahim Khalaj, and Marwan Abi-Antoun. Comparative evaluation of architectural and code-level approaches for finding security vulnerabilities. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 27–34, New York, NY, USA, 2014. ACM.

[64] Marco Vieira, Nuno Antunes, and Henrique Madeira. Using web security scanners to detect vulnerabilities in web services. *2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009.

[65] Ju An Wang, Minzhe Guo, Hao Wang, Min Xia, and Linfeng Zhou. Environmental metrics for software security based on a vulnerability ontology. In *Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '09, pages 159–168, Washington, DC, USA, 2009. IEEE Computer Society.

[66] Ju An Wang, Minzhe Guo, Hao Wang, Min Xia, and Linfeng Zhou. Ontology-based security assessment for software products. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 15:1–15:4, New York, NY, USA, 2009. ACM.

[67] Ju An Wang, Minzhe Guo, Hao Wang, and Linfeng Zhou. Measuring and ranking attacks based on vulnerability analysis. *Inf. Syst. E-bus. Manag.*, 10(4):455–490, December 2012.

[68] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[69] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 209–218, New York, NY, USA, 2012. ACM.

[70] Shoji YUEN, Keishi KATO, Daiju KATO, and Kiyoshi AGUSA. Web automata: A behavioral model of web applications based on the mvc model. *Computer Software*, 22(2):44–57, 2005.

付 録A　BSIMMの分類

表 A.1: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (GOVERNANCE: STRATEGY AND METRICS)

	Objective	Activity	Level	
SM1.1	make the plan explicit	publish process (roles, responsibilities, plan), evolve as necessary	1	開発手法非依存
SM1.2	build support throughout organization	create evangelism role and perform internal marketing	1	開発手法非依存
SM1.3	secure executive buy-in	educate executives	1	開発手法非依存
SM1.4	establish SSDL gates (but do not enforce)	identify gate locations, gather necessary artifacts	1	非整合 (プロセス)
SM1.6	make clear who's taking the risk	require security sign-off	1	開発手法非依存
SM2.1	foster transparency (or competition)	publish data about software security internally	2	開発手法非依存
SM2.2	change behavior	enforce gates with measurements and track exceptions	2	非整合 (プロセス)
SM2.3	create broad base of support	create or grow a satellite	2	開発手法非依存
SM2.5	define success	identify metrics and use them to drive budgets	2	開発手法非依存
SM3.1	know where all apps in your inventory stand	use an internal tracking application with portfolio view	3	開発手法非依存
SM3.2	create external support	run an external marketing program	3	開発手法非依存

表 A.2: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (GOVERNANCE: COMPLIANCE AND POLICY)

	Objective	Activity	Level	
CP1.1	understand compliance drivers (FFIEC, GLBA, OCC, PCI, SOX, HIPAA)	unify regulatory pressures	1	開発手法非依存
CP1.2	promote privacy	identify PII obligations	1	開発手法非依存
CP1.3	meet regulatory needs or customer demand with a unified approach	create policy	1	開発手法非依存
CP2.1	promote privacy	identify PII data inventory	2	開発手法非依存
CP2.2	ensure accountability for software risk	require security sign-off for compliance-related risk	2	開発手法非依存
CP2.3	align practices with compliance	implement and track controls for compliance	2	開発手法非依存
CP2.4	ensure vendors don't screw up compliance	paper all vendor contracts with software security SLAs	2	開発手法非依存
CP2.5	gain executive buy-in	promote executive awareness of compliance and privacy obligations	2	開発手法非依存
CP3.1	demonstrate compliance story	create regulator eye-candy	3	開発手法非依存
CP3.2	manage third-party vendors	impose policy on vendors	3	開発手法非依存
CP3.3	keep policy aligned with reality	drive feedback from SSDL data back to policy (T: strategy/metrics)	3	開発手法非依存

表 A.3: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (GOVERNANCE: TRAINING)

	Objective	Activity	Level	
T1.1	promote culture of security throughout the organization	provide awareness training	1	開発手法非依存
T1.5	build capabilities beyond awareness	deliver role-specific advanced curriculum (tools, technology stacks, bug parade)	1	開発手法非依存
T1.6	see yourself in the problem	create and use material specific to company history	1	整合、自動化
T1.7	reduce impact on training targets and build delivery staff	deliver on-demand individual training	1	開発手法非依存
T2.5	educate/strengthen social network	enhance satellite through training	2	開発手法非依存
T2.6	ensure new hires enhance culture	include security resources in onboarding	2	開発手法非依存
T2.7	create social network tied into dev	identify satellite through training	2	開発手法非依存
T3.1	align security culture with career path	reward progression through curriculum (certification or HR)	3	開発手法非依存
T3.2	spread security culture to providers	provide training for vendors or outsource workers	3	開発手法非依存
T3.3	market security culture as differentiator	host external software security events	3	開発手法非依存
T3.4	keep staff up-to-date and address turnover	require annual refresher	3	開発手法非依存
T3.5	act as informal resource to leverage teachable moments	establish SSG office hours	3	非整合

表 A.4: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (INTELLIGENCE: ATTACK MODELS)

	Objective	Activity	Level	
AM1.1	understand attack basics	build and maintain a top N possible attacks list	1	開発手法非依存
AM1.2	prioritize applications by data consumed/manipulated	create data classification scheme and inventory	1	非整合 (文書化)
AM1.3	understand the "who" of attacks	identify potential attackers	1	非整合 (文書化)
AM1.4	understand the organization's history	collect and publish attack stories	1	非整合 (文書化)
AM1.5	stay current on attack/vulnerability environment	gather attack intelligence	1	非整合 (専門性)
AM1.6	communicate attacker perspective	build an internal forum to discuss attacks (T: standards/req)	1	開発手法非依存 (整合)
AM2.1	provide resources for security testing and AA	build attack patterns and abuse cases tied to potential attackers	2	非整合 (文書化、専門性)
AM2.2	understand technology-driven attacks	create technology-specific attack patterns	2	開発手法非依存
AM3.1	get ahead of the attack curve	have a science team that develops new attack methods	3	非整合 (専門性)
AM3.2	arm testers and auditors	create and use automation to do what the attackers will do	3	非整合 (専門性) 自動化

表 A.5: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (INTELLIGENCE: SECURITY FEATURES AND DESIGN)

	Objective	Activity	Level	
SFD1.1	create proactive security guidance around security features	build and publish security features	1	非整合 (文書化)
SFD1.2	inject security thinking into architecture group	engage SSG with architecture	1	非整合
SFD2.1	create proactive security design based on technology stacks	build secure-by-design middleware frameworks and common libraries (T: code review)	2	開発手法非依存 整合
SFD2.2	address the need for new architecture	create SSG capability to solve difficult design problems	2	非整合
SFD3.1	formalize consensus on design	form review board or central committee to approve and maintain secure design patterns	3	非整合 (組織)
SFD3.2	promote design efficiency	require use of approved security features and frameworks (T: AA)	3	開発手法非依存
SFD3.3	practice reuse	find and publish mature design patterns from the organization	3	非整合 (文書化)

表 A.6: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (INTELLIGENCE: STANDARDS AND REQUIREMENTS)

	Objective	Activity	Level	
SR1.1	meet demand for security features	create security standards (T: sec features/design)	1	開発手法非依存
SR1.2	ensure that everybody knows where to get latest and greatest	create security portal	1	開発手法非依存 (自動化)
SR1.3	compliance strategy	translate compliance constraints to requirements	1	開発手法非依存
SR1.4	tell people what to look for in code review	use secure coding standards	1	整合
SR2.2	formalize standards process	create a standards review board	2	非整合 (組織)
SR2.3	reduce SSG workload	create standards for technology stacks	2	開発手法非依存
SR2.4	manage open source risk	identify open source	2	開発手法非依存
SR2.5	gain buy-in from legal department and standardize approach	create SLA boilerplate (T: compliance and policy)	2	開発手法非依存
SR3.1	manage open source risk	control open source risk	3	開発手法非依存
SR3.2	educate third-party vendors	communicate standards to vendors	3	開発手法非依存

表 A.7: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (SSDL TOUCHPOINTS: ARCHITECTURE ANALYSIS)

	Objective	Activity	Level	
AA1.1	get started with AA	perform security feature review	1	整合 (専門家必要)
AA1.2	demonstrate value of AA with real data	perform design review for high-risk applications	1	整合 (専門家必要)
AA1.3	build internal capability on security architecture	have SSG lead review efforts	1	非整合 (第三者 SSG)
AA1.4	have a lightweight approach to risk classification and prioritization	use a risk questionnaire to rank applications	1	整合 (専門家必要)
AA2.1	model objects	define and use AA process	2	自動化
AA2.2	promote a common language for describing architecture	standardize architectural descriptions (including data flow)	2	非整合 (文書化)
AA2.3	build capability organization-wide	make SSG available as AA resource or mentor	2	非整合
AA3.1	build capabilities organization-wide	have software architects lead review efforts	3	非整合
AA3.2	build proactive security architecture	drive analysis results into standard architectural patterns (T: sec features/design)	3	非整合

表 A.8: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (SSDL TOUCHPOINTS: CODE REVIEW)

	Objective	Activity	Level	
CR1.1	know which bugs matter to you	create top N bugs list (real data preferred) (T: training)	1	開発手法非依存
CR1.2	review high-risk applications opportunistically	have SSG perform ad hoc review	1	非整合 (第三者 SSG)
CR1.4	drive efficiency/consistency with automation	use automated tools along with manual review	1	自動化
CR1.5	find bugs earlier	make code review mandatory for all projects	1	開発手法非依存
CR1.6	know which bugs matter (for training)	use centralized reporting to close the knowledge loop and drive training (T: strategy/metrics)	2	自動化
CR2.2	drive behavior objectively	enforce coding standards	2	整合
CR2.5	make most efficient use of tools	assign tool mentors	2	整合
CR2.6	drive efficiency/reduce false positives	use automated tools with tailored rules	2	整合（自動化）
CR3.2	combine assessment techniques	build a factory	3	開発手法非依存
CR3.3	handle new bug classes in an already scanned codebase	build capability for eradicating specific bugs from entire codebase	3	整合
CR3.4	address insider threat from development	automate malicious code detection	3	開発手法非依存

表 A.9: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (SSDL TOUCHPOINTS: SECURITY TESTING)

	Objective	Activity	Level	
ST1.1	execute adversarial tests beyond functional	ensure QA supports edge/boundary value condition testing	1	非整合 (QA が第三者の場合)
ST1.3	start security testing in familiar functional territory	drive tests with security requirements and security features	1	(半) 自動化
ST2.1	use encapsulated attacker perspective	integrate black box security tools into the QA process	2	非整合 (QA が第三者の場合)
ST2.4	facilitate security mindset	share security results with QA	2	開発手法非依存
ST3.1	include security testing in regression	include security tests in QA automation	3	自動化
ST3.2	teach tools about your code	perform fuzz testing customized to application APIs	3	自動化
ST3.3	probe risk claims directly	drive tests with risk analysis results	3	開発手法非依存
ST3.4	drive testing depth	leverage coverage analysis	3	自動化
ST3.5	move beyond functional testing to attacker’s perspective	begin to build and apply adversarial security tests (abuse cases)	3	開発手法非依存

表 A.10: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (DEPLOYMENT: PENETRATION TESTING)

	Objective	Activity	Level	
PT1.1	demonstrate that your organization's code needs help too	use external penetration testers to find problems	1	非整合 (QA が第三者の場合)
PT1.2	fix what you find to show real progress	feed results to the defect management and mitigation system (T: config/vuln mgmt)	1	自動化
PT1.3	create internal capability	use penetration testing tools internally	1	自動化
PT2.2	promote deeper analysis	provide penentration testers with all available information (T: AA and code review)	2	非整合 (QA が第三者の場合)
PT2.3	sanity check constantly	schedule periodic pen tests for application coverage	2	自動化
PT3.1	keep up with edge of attacker's perspective	use external penetration testers to perform deep-dive analysis	3	非整合 (QA が第三者の場合)
PT3.2	automate for efficiency without losing depth	have the SSG customize penetration testing tools and scripts	2	自動化

表 A.11: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (DEPLOYMENT: SOFTWARE ENVIRONMENT)

	Objective	Activity	Level	
SE1.1	watch software	use application input monitoring	1	開発手法非依存
SE1.2	provide a solid host/network foundation for software	ensure host and network security basics are in place	1	開発手法非依存
SE2.2	guide operations on application needs	publish installation guides	2	開発手法非依存
SE2.4	protect apps (or parts of apps) that are published over trust boundaries	use code signing	2	開発手法非依存
SE3.2	protect IP and make exploit development harder	use code protection	3	開発手法非依存
SE3.3	watch software	use application behavior monitoring and diagnostics	3	開発手法非依存

表 A.12: BSIMM-V のセキュリティ保証手法とアジャイルソフトウェア開発との整合性 (DEPLOYMENT: CONFIGURATION MANAGEMENT AND VULNERABILITY MANAGEMENT)

	Objective	Activity	Level	
CMVM1.1	know what to do when something bad happens	create or interface with incident response	1	開発手法非依存
CMVM1.2	use ops data to change dev behavior	identify software defects found in operations monitoring and feed them back to development	1	開発手法非依存
CMVM2.1	be able to fix apps when they are under direct attack	have emergency codebase response	2	開発手法非依存
CMVM2.2	use ops data to change dev behavior	track software bugs found during ops through the fix process	2	開発手法非依存
CMVM2.3	know where the code is	develop an operations inventory of applications	2	開発手法非依存
CMVM3.1	learn from operational experience	fix all occurrences of software bugs found in operations (T: code review)	3	開発手法非依存
CMVM3.2	use ops data to change dev behavior	enhance the SSDL to prevent software bugs found in operations	3	開発手法非依存
CMVM3.3	ensure processes are in place to minimize software incident impact	simulate software crisis	3	開発手法非依存
CMVM3.4	engage external researchers in vulnerability discovery	operate a bug bounty program	3	非整合 (第三者)

アジャイル Web アプリケーション開発におけるソフトウェアセキュリティ保証のための反復型評価手法に関する研究

著者 宗藤 誠治
発行 2016 年 3 月