IPA

セキュア･プログラミング講座

IPA 独立行政法人情報処理推進機構
セキュリティセンター

目次

はじめに 1
本書の対象読者 2
注意事項 2
1. 脆弱性 3
1.1. 既知の脆弱性への対応 4
1.2. 未知と対応漏れを意識した対応 6
2. 原則 7
2.1. 設計原則 7
2.2. 実装原則 12
3. 脅威モデリング 17
3.1. 脅威の分類：STRIDE 18
3.2. 脅威モデリングの手順 19
4. 開発プロセス 21
5. 計画すべきセキュリティ機能 27
5.1. 認証 28
5.2. 認可 28
5.3. 暗号化 29
5.4. 入力の検証 29
5.5. 出力のエスケープ（無害化） 30
5.6. 例外処理 30
5.7. ロギング 31
おわりに 32
付録 33
付録 A:Web アプリケーション編に関して 34
付録 B:C/C++言語編に関して 36

はじめに

これまでのセキュア・プログラミング講座では、2002 年 2 月に「Web プログラマコース」と「製品プログラマコース」、2007 年の 6 月に「Web アプリケーション編」と同年の 9 月に「C/C++編」を公開した。

2002 年の最初の版の製品プログラマコースの第 10 章「より堅固なソフトウェア構築」の「設計段階からのセキュリティ」で上流工程からセキュリティに取り組むことを訴えた。2007 年の版では Web アプリケーション編、C/C++言語編の両編ともに「開発工程と脆弱性対策」として要件定義工程からセキュア･プログラミングに取り組むべきことを明確にし、更に「C/C++言語編」では　脅威モデリング、セキュリティレビュー、ソースコードレビュー、セキュリティテストや暗号化なども取り上げた。

今回は原則を中心として共通的なものをまとめて再編することにした。個別の脆弱性対策等の部分は、本講座以外にも、「安全なウェブサイトの作り方」や IPA　SEC の組込み系ソフトウェア開発関連の C および C++、JPCERT/CC のセキュアコーディングなどで適宜更新がされているので、そちらを参照することにして、付録として取り扱うことにした。

リスク＝【情報資産】×【脅威】×【脆弱性】

リスクは損害が発生する可能性であり、損害の発生を抑えるためにリスクを低くしようとする。この式の考え方では【情報資産】、【脅威】、【脆弱性】のどれかが無くなればリスク無いことになり、損害が発生することはないことになる。

セキュア・プログラミングには脆弱性を作りこまない根本的解決策と予防的に影響を軽減する保険的対策あるが、本書では両方に対しての対策を記載する。

本書の構成

第 1 章 脆弱性では、定義と既に知られているものとそれ以外への取り組み方に関して説明する。
第 2 章 原則では、脆弱性を作りこまないために設計時と実装時のそれぞれで意識すべき原則について説明する。
第 3 章 脅威モデリングでは、どこに集中すべきかを判断するための手順について説明する。
第 4 章 開発プロセスでは、上流工程からの作り込みの実施と作り込んだものを確認するレビューやテストなどを説明する。
第 5 章 計画すべきセキュリティ機能では、セキュアなシステムを構築する際に一般的に必要になるセキュリティ機能に関して説明する。

本書の対象読者

・広い意味でのソフトウェア開発者：ソフトウェアを開発するプロジェクトの関係者（プロジェクトマネージャ、要件定義者等も含む）特にプロジェクト内の標準作成に携わるアーキテクトとプログラマ

注意事項

・本講座内で参照している URL に関しては、文書作成時のものを脚注に記載した。URL は変更されることがあるので、できるだけ当講座の案内ページ[1]で情報提供する。

[1] https://www.ipa.go.jp/security/awareness/vendor/programming/index.html

1. 脆弱性（ぜいじゃくせい）

本章では、本講座で対象とする脆弱性は、根本的解決策を取れる既知のものとそれ以外の両方に関して取り扱う。

2007 年に公開した本講座では、「脆弱性とは、想定した用途以外の動作をするように攻略されてしまう可能性であるといえる。」としてきた。

総務省の国民のための情報セキュリティサイト[2]では、「脆弱性とは、コンピュータのOS やソフトウェアにおいて、プログラムの不具合や設計上のミスが原因となって発生した情報セキュリティ上の欠陥のこととされている。」と定義している。

「情報セキュリティ早期警戒パートナーシップガイドライン[3]」では、「脆弱性とは、ソフトウェア製品やウェブアプリケーション等において、コンピュータ不正アクセスやコンピュータウィルス等の攻撃により、その機能や性能を損なう原因となり得るセキュリティ上の問題箇所です。」と定義している。

脆弱性とは、プログラムや設定上の問題に起因するセキュリティ上の「弱点」である。

脆弱性が残された状態でコンピュータを利用していると、侵入されたり、不正アクセスに利用されたり、ウイルスに感染したりする危険性がある。

本講座では、基本として設定上の問題やミスに関しては取り扱わない。しかしながら、設定を変更するプログラムを作成する場合にはこの限りではない。

[2] http://www.soumu.go.jp/main_sosiki/joho_tsusin/security/basic/risk/11.html
[3] https://www.ipa.go.jp/security/ciadr/partnership_guide.html

1.1. 既知の脆弱性への対応

個々のアプリケーションの既知の脆弱性に関しては、開発者も必ずしも十分に認識できていないことがある。従来のようにアプリケーションが実際に稼動させてテストするよりもずっと前でも、アプリケーションが使用するプログラミング言語や更に使用する環境にはそれぞれ既に知られている脆弱性は存在している。これは例えば DBMS で入力値を用いて SQL 文の文字列を生成している場合ならば SQL インジェクションが既知の脆弱性に含まれる。

本来は既知の脆弱性に対して、判明している脆弱性全てに根本的対策を行うべきだと考えるところだが、必ずしも全ての既知の脆弱性に対して全て根本的対策を行わなければならないわけではない。これは、例えば自分で作成したプログラムを今 1 度だけ実行する際に、読み込むみデータが 1 万行もないことが明白なのにプログラム上の整数値のオーバーフロー問題が起きるかもしれないからとプログラムを本当に修正する必要があるかということになる。このような例以外にも HW に実装されているメモリの容量など実行環境によって実行できなくなることなどもある。これは例えば昔町の小さな八百屋がつり銭を笊に入れているのをお金だから、必ず金庫で管理しなければならないし、せっかく金庫を使うならば、銀行の金庫と同様に安心なものでなければならないと言っているようなものである。

製品ソフトウェアの脆弱性を日々収録している Web サイトがインターネット上にはある。悪用を防ぐために、詳細な実装ミスの内容や攻撃手口は明かされない場合が多いが、脆弱性の内容確認や対策の実施判断などに役立てることができる。それらの例を掲げる。

- **脆弱性関連情報に関する届出状況（IPA）**[4]
  IPA に届け出られた脆弱性の概要と統計情報である。経済産業省告示「ソフトウェア等脆弱性関連情報取扱基準」にもとづいて運営されている届出制度への届出状況である。

- **JVN（Japan Vulnerability Notes）**[5]
  「情報セキュリティ早期警戒パートナーシップ」に基いて IPA と JPCERT/CC が取り扱った脆弱性情報を公開している。枠組みに参加している日本国内の製品開発者の対応状況も含まれている。対応状況には、脆弱性に該当する製品の有無、回避策や対策情報（パッチ等）も含まれる。このサイトには米国 US-CERT、CERT/CC お

[4] https://www.ipa.go.jp/security/vuln/report/press.html

[5] https://jvn.jp/

よび英国 CPNI (NISCC)からの情報も掲載されている。JVN も経済産業省告示「ソフトウェア等脆弱性関連情報取扱基準」に基づいて運営されている。

- **CVE（Common Vulnerabilities and Exposures）**[6]
  複数の組織から発表される脆弱性情報に共通の識別番号（CVE 番号）を与えるプロジェクトである。

- **CWE（Common Weakness Enumeration）**[7]
  製品の個々の脆弱性ではなく、ソフトウェアの弱点（脆弱性や不具合）のパターンに共通の呼び名を与えるプロジェクトである。

- **SecurityFocus**[8]
  ソフトウェア製品に検出された脆弱性、攻撃手口と防御手法、セキュリティ・ツール等を収集、掲載している。

特に IPA が提供している情報に関しては IPA テクニカルウォッチ「脆弱性対策の効果的な進め方（実践編）[9]」や「脆弱性対策コンテンツリファレンス[10]」でより詳しく紹介している。

---

[6] http://cve.mitre.org/

[7] http://cwe.mitre.org/

[8] http://www.securityfocus.com/

[9] https://www.ipa.go.jp/security/technicalwatch/20150331.html

[10] https://www.ipa.go.jp/files/000051352.pdf

1.2. 未知と対応漏れを意識した対応

脆弱性は既知のものだけではなく未知のものもある。未知の脆弱性に対応をと言われても、未知の脆弱性への対応をできるものではない。

セキュリティ対策すべき資産を特定し、特定した資産に対しての脅威を分析し、その脅威に関して脆弱性があるものとしてリスクを算出し、その必要性に応じてなんらかの対策を取ることになる。しかしながら、既知の脆弱性への根本的な対策とは異なり、必ずしも具体的な対応箇所が明確になるわけではないこともあり、普通は影響を軽減することにとどまってしまう。更にその対策の範囲は根本的な対策に比べて広くなる傾向がある。可能性をある程度低減したり、被害範囲をある程度狭くしたり、早期に気づけるようになるだけだと割り切って行うことになる。それでも何もやらないよりも十分に効果があると割り切って行う必要がある。

対応漏れも未知と同様に考える必要がある。漏れなくやれればよいが実際には発生してしまう。漏れが発生することはあるものと考えて「安全なウェブサイトの作り方」にも示している保険的対策をしておくことが必要になる。

2. 原則

本章では、脆弱性を作りこまないためのセキュア･プログラミングでの原則を、設計時と実装時に分けて取り扱う。

JIS Q 27002 (ISO/IEC 27002) では、情報セキュリティを「情報の機密性、完全性および可用性を維持すること。さらに、真正性、責任追跡性、否認防止および信頼性のような特性を維持することを含めてもよい。」と定義している。すなわち、セキュアであるとは、機密性(confidentiality)、完全性 (integrity)、可用性 (availability)を維持できていることであり、更に真正性 (authenticity)、責任追跡性 (accountability)、否認防止 (non-repudiation)や信頼性 (reliability)の特性も維持できている状態である。情報システムでは機密性を重視して CIA と表現されるが、制御システムでは可用性を重視して AIC と表現されることがある。

個々の事象をすべて明確にして、その対策を挙げるようなことは現実的にできるものではない。このような場合に原則を定めてそれに従うことで、必ずしも個別に明確に決められていない場合にも対応できるように準備することができる。原則はできれば十分少ない数で、更にあまり頻繁に変える必要のないものであることがのぞましい。

コラム：

インターネット上で「security principles」を検索すると多くの情報が得られる。

参考として例えば次のものは参照してみると良い。

SANS の Principles of secure design [11]

Build Security In の Design Principles [12]

Oracle linux の Design Principles for Secure Coding [13]

OWASP Secure Coding Principles [14]

NIST　SP800-27 [15]

IATAC Software Security Assurance [16]

[11] https://www.sans.org/reading-room/whitepapers/application/secure-design-exploit-infusion-35587
[12] https://buildsecurityin.us-cert.gov/articles/knowledge/principles/design-principles
[13] https://docs.oracle.com/cd/E37670_01/E36387/html/ol_desprinsc_sec.html
[14] https://www.owasp.org/index.php/Secure_Coding_Principles
[15] http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf
[16] http://www.dtic.mil/dtic/tr/fulltext/u2/a472363.pdf

2.1. 設計原則

設計原則には、多くの設計原則でも参照されている 1975 年の古典ともいえる Saltzer & Schroeder の 8 原則[17] を取り上げる。

１．**Economy of mechanism：効率的なメカニズム**
単純で小さな設計を心がける。

２．**Fail-safe defaults：フェイルセーフなデフォルト**
必要ないものを排除するのではなく、必要なものを許す判断を基本とする。

３．**Complete mediation：完全な仲介**
あらゆるオブジェクトに対するすべての処理に関与する。

４．**Open design：オープンな設計**
設計内容を秘密していることに頼らない。

５．**Separation of privilege：権限の分離**
1 つの鍵を持つ者にアクセスを許してしまう仕組みよりも複数の鍵を使って保護する方が、強固だし柔軟でもある。

６．**Least privilege：最小限の権限**
システムのすべてのプログラムおよびすべてのユーザは、業務を遂行するために必要な最小の権限の組み合わせを使って操作を行うべきである。

７．**Least common mechanism：共通メカニズムの最小化**
複数のユーザが共有し依存する仕組みの規模を最小限に押える。

８．**Psychological acceptability：心理学的受容性**
ユーザが日常的に無意識のうちに保護の仕組みを正しく利用できるように、使いやすさを優先した設計が重要である。

本頁の原則の訳は「C/C++セキュアコーディング[18]」の「8.1 安全なソフトウェア開発のための原則」から引用した。

[17] http://web.mit.edu/Saltzer/www/publications/protection/Basic.html
[18] ·Robert C. Seacord 著, 歌代和正監訳 JPCERT コーディネーションセンター訳, 『C/C++セキュアコーディング』, KADOKAWA/アスキー・メディアワークス （2006 年 11 月）

設計原則1：Economy of mechanism　効率的なメカニズム

単純で小さな設計を心がけること。

仕組みを単純に。　システムは小さく単純明快に設計する。

「KISS」の原則" Keep it short and simple”または”keep it simple, stupid”である。

　メカニズムが経済的であること。　システムは小さく単純に素直に設計すべきである。

大きく複雑なものは間違いやすく、それ故に経済的でなくなる。

設計原則2：Fail-safe defaults　フェイルセーフなデフォルト

誤操作を安全に処理する。誤操作・誤動作による障害が発生した場合でも常に安全側に制御するというフェイルセーフな考え方をする。

必要ないものを排除するのではなく、必要なものを許すという判断を基本とする。

これは禁止を基本とすることで、明示的許可されない限りはデフォルトとしてサービスを拒否すべきというものである。

設計原則3：Complete mediation　完全な仲介

完全に仲介を行う。セキュティメカニズムが漏れなく適用されるようにする。

すなわち用意したセキュリティ機能がバイパスされないようにする。

考え得るアクセス方法は全てチェックしなければならない。チェックメカニズムは 回避できないように配置すること必要である。 例えば、クライアント-サーバモデルにおいては、独自のクライアントを作ることができるのでクライアント側でのチェックでは不十分でサーバ側で全てのアクセスをチェックしなければならない。

　CERT Top 10 Secure Coding Practices の Bonus Photograph[19]にこの原則のアンチパターンとなる、たとえゲートを用意しても回避できると意味がないことを示した写真があるので参照してみると良い。

---

[19] この写真の著作権が不明のためリンク情報のみとする。
https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices#Top10SecureCodingPractices-BonusPhotograph

設計原則4：Open design オープンな設計

設計内容を秘密にしてはならない。(設計を隠すことで守れるとはしない)
･オープンな設計であること。 防御メカニズムは「攻撃者がメカニズムの仕組みを知らないこと」をあてにしてはならない。 逆に防御メカニズムそのものの内容は公開されているべきで、メカニズムの安全性は パスワード等、比較的少数のしかも容易に変更可能な要素に依存すべきである。そうすれば、数多くの 第三者の検証を受けることが可能となる。
広く配布したシステムを密かにメンテナンスしようとすることは、 現実的ではない。 逆コンパイラやハードウェアを壊してしまうことで、短時間にあらゆる「秘密」が明らかになってしまう可能性がある。

設計原則5：Separation of privilege 権限の分離

権限を分離する。
可能であれば、1 つの鍵で保護する仕組みより 2 つの鍵を使って保護する方が強固だし柔軟でもある。
理想的には、アクセスは複数の条件に基づいて行うべきである。そうすれば、一つの防御メカニズムが破られても完全なアクセスを許すことはない。
多層防御（Defense in depth）や責務の分離（Separation of duty）につながる原則である。

設計原則6：Least privilege 最小限の権限

特権をできるだけ持たせない。
システムのすべてのプログラムおよびすべてのユーザは、業務を遂行するために必要な最小の権限の組み合わせを使って操作を行うべきである。
そうすれば、攻撃を受けたときの被害を最小限に抑えることができる。

設計原則7：Least common mechanism 共通メカニズムの最小化

複数のユーザが共有し依存する仕組みの規模を最小限に押える。
共通して持つメカニズムを最小限にすること。 共有されているオブジェクト(たとえば、/tmp や /var/tmp の利用)は情報流出の経路となる危険性を持っている。この場所で予期しない相互作用が発生する恐れがある。

設計原則8：Psychological acceptability　心理学的受容性

ユーザおよび開発者が受け入れられる、簡単に使えるように設計する。使いやすい仕組みはユーザに避けられ難く、使い難い仕組みはユーザや開発者が避けたがる。ユーザが日常的に無意識のうちに保護の仕組みを正しく利用できるように、使いやすさを優先した設計が重要である。

セキュリティの仕組みがユーザ思い描く防御の目標と合っているならば、過ちは減るだろう。

2.2. 実装原則

実装原則には SEI　CERT　Top 10 Secure Coding Practices[20]を取り上げることにする。但し、ここでは原文で 7 番目にあった Sanitize data sent to other systems.は根本的解決策として重要であり、1 番目の入力と対に意識してもらいたいので、あえて 2 番目にした。

1.Validate input. ※1
2.Sanitize data sent to other systems.
3.Heed compiler warnings.
4.Architect and design for security policies.
5.Keep it simple. ※2
6.Default deny. ※2
7.Adhere to the principle of least privilege. ※2
8.Practice defense in depth. ※1
9.Use effective quality assurance techniques. ※1
10.Adopt a secure coding standard.

※1[Seacord, R. *Secure Coding in C and C++*. Upper Saddle River, NJ: Addison-Wesley, 2006 (ISBN 0321335724) [21]]

※2[Saltzer, J. H. "Protection and the Control of Information Sharing in Multics." *Communications of the ACM 17*, 7 (July 1974): 388-402. [22]、
Saltzer, J. H. & Schroeder, M. D. "The Protection of Information in Computer Systems." *Proceedings of the IEEE 63*, 9 (September 1975), 1278-1308. [23]]

---

[20]https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices

[21] http://resources.sei.cmu.edu/asset_files/BookChapter/2005_009_001_52692.pdf
[22] https://cseweb.ucsd.edu/classes/wi08/cse221/papers/saltzer74.pdf
[23] http://web.mit.edu/Saltzer/www/publications/protection/Basic.html

実装原則1：Validate input.

すべての信頼されていないデータソースからの入力を検証する。

適切な入力検証は、多くのソフトウェアの脆弱性を緩和することができる。

コマンドライン引数、ネットワーク・インタフェース、環境変数、およびユーザが管理しているファイルなどほとんどの外部データソースは信頼できない。

DBMS から取得したものも行うこと。特に Web の場合はユーザフォームからの入力だけではなく、クッキーや HTML のヘッダーブロックの値なども含めてクライアントから受け取った値を使用する場合には精査する。

実装原則2：Sanitize data sent to other systems.

外部に渡すデータは渡した先で問題を起こさないように加工する。渡す先によって問題となる条件は異なるのでそれに合わせた加工をする必要がある。例えばデータを渡す先が Web の HTML の BODY 部分ならば XSS 対策をすることが必要になるし、DBMS の SQL ならば SQL インジェクション対策を行う必要がある。

コマンドシェル、リレーショナルデータベース、および商用オフザシェルフ（COTS）コンポーネントのような複雑なサブシステムに渡されるすべてのデータを無害化する必要がある。

[C STR02-A[24]]

不正文字の無害化以外にもデータを渡した先で問題を起こさないようにするためには、データ型に応じたデータの取り扱いや本来利用すべきでない関数やライブラリの利用などにも注意する必要がある。

攻撃者は SQL、コマンドまたほかのインジェクション攻撃を通して、これらのコンポーネントで通常使用されていない機能を呼び出すことができるかもしれない。

呼び出された複雑なサブシステムは呼び出しが行われるコンテキストを理解していないので、この問題は必ずしも入力検証の問題ではなく、出力側の問題である。なぜなら呼び出し元プロセスはコンテキストを理解しているので、サブシステムなどを起動する前にデータを無害化することができ、その責任がある。

---

[24] https://www.securecoding.cert.org/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems
https://www.jpcert.or.jp/sc-rules/c-str02-c.html

実装原則3：Heed compiler warnings.

コンパイラの警告に注意を払わなければならない。

コンパイラはプログラムコードに対して必ず行われる最初の精査行為である。

コンパイラのオプションを設定することでより多くの情報を得ることができる。

[C MSC00-A[25]、C ++ MSC00-A[26]]

コンパイラからの警告は有効な情報が提供されているはずである。しかしながら、開発者は警告レベルだと　そのままでも実行可能だったり、非常に多くの同様のものが出力されたりするので無視してしまう場合がある。

可能ならばコンパイラだけではなく、さらに静的および動的解析ツールを使用することで、より多くのセキュリティ上の欠陥を検出でき、それを排除することができる。

実装原則4：Architect and design for security policies.

セキュリティポリシー実現のための実装と設計を行う。

守るべきものを特定してそれを守るために行うものであり、すべてを同様に守るようにするものではない。極端な表現ではあるが、すべてを同様に守るということはすべてを同様に守らないということと同じ意味になる。

それぞれのアプリケーションやシステムで決めたセキュリティポリシーにしたがって行う。ソフトウェアアーキテクチャを作成し、実装し、セキュリティポリシーを適用するためのソフトウェアを設計する。

---

25 https://www.securecoding.cert.org/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels
https://www.jpcert.or.jp/sc-rules/c-msc00-c.html

26 https://www.securecoding.cert.org/confluence/display/cplusplus/VOID+Compile+cleanly+at+high+warning+levels

実装原則5：Keep it simple.

シンプルを維持する。設計原則１の「Economy of mechanism：効率的なメカニズム」と同じ意図のものである。同じ結果を得られる実装は、ひとつとは限らず、複数の選択候補があるならば、できるだけシンプルなものを選択すべきである。シンプルにすることで最初の開発からその後のデバッグや保守といった開発作業全般でミスを犯す可能性を低くできる。逆に複雑にしても攻撃を避けられるわけではなく単にミスの発生を高め、それが結果的に脆弱性となる可能性を高めている。

可能な限りシンプルで小さなデザインにする。

複雑な設計は、実装中、構成中および使用中にエラーとなる可能性を高める。更に、複雑な設計はセキュリティメカニズムが複雑になり、適切なレベルを達成するために必要な労力は劇的に増加することになる。

実装原則6：Default deny.

拒否をデフォルトにする。設計原則２の「2.Fail-safe defaults：フェイルセーフなデフォルト」と同じ意図のものである。

許可ベースではなく、拒否ベースでアクセス決定する。

デフォルトではアクセスが拒否され、保護スキームはアクセスが許可される条件を識別していることを意味する。

Order Allow,Deny のホワイトリスト方式と表現されることと同じ意図のものである。

実装原則7：Adhere to the principle of least privilege.

最小権限の原則に従う。設計原則６の「Least privilege：最小限の権限」と同じものである。

どのプロセスも、実行するために必要な最低限の特権セットで実行すべきである。

権限が昇格されている時間を最小限にするべきである。このアプローチによって、攻撃者が昇格した権限で任意のコードを実行する機会を減らすことができる。

実装原則8：Practice defense in depth.

多層防御を行う。設計原則５の「Separation of privilege：権限の分離」と同じ意図のものである。根本的対策だけでなく、保険的対策も含めた異なるタイプ防御策を行うようにすることである。すなわちひとつの対策がもしも不完全であったり、攻撃者に破られたりとしても全てを失ってしまうのではなく、被害がある程度限定できるようにする。これは例えば潜水艦での水密隔壁のように、たとえ浸水が起きても全部ではなく一部だけに限定することで沈没を避けるようにすることである。
複数の防御的な戦略でリスクを管理する。これは防衛の１つの層が不十分である場合でも、別の層の防衛が悪用可能な脆弱性をセキュリティ欠陥にならないようにしたり、例え脆弱性が悪用された場合でも影響を制限したりする。
例えば、セキュアな実行環境とセキュアなプログラミング技術を組み合わせることで、デプロイメント時にコードに残っている運用環境に攻撃できる脆弱性の可能性を減らせる可能性がある。

実装原則9：Use effective quality assurance techniques.

効果的な品質保証テクニックを使う。
優れた品質保証技術は、脆弱性を特定し、排除するのにも有効である。
Fuzz テスト、侵入テスト、およびソースコードの監査は、すべての効果的な品質保証プログラムの一部として組み込まれるべきである。
独立したセキュリティレビューは、よりセキュアなシステムにつながることができる。
例えば、外部のレビュアーは独立の立場で無効な前提条件を特定され修正をもたらす。

実装原則10：Adopt a secure coding standard.

セキュアコーディング標準を採用する。
ターゲット開発言語やプラットフォームのためのセキュアコーディング標準を適用し開発する。
多くのセキュリティ対策は複数の箇所で同様の対策実施を行うことになる。標準を採用して共通的に対応することで効率的に対応することができる。
　もしも採用した標準で対策が不十分な部分が見つかったとしても、同様に対応できるので比較的簡単に修正することができる。

3. 脅威モデリング

本章では、どこに集中すべきかを判断するための手順として脅威モデリングを取り扱う。
　本書のはじめに　でも取り上げた　リスク＝【情報資産】×【脅威】×【脆弱性】に従ってより高いリスクに対して優先的に対策を実施していくために脅威モデリングを行う。

脅威モデリングを行おう

脅威モデリングは、開発対象のソフトウェアがどのようなセキュリティ脅威にさらされており、攻略される可能性を持ちうるかを洗い出す活動のひとつである。脅威モデリングはセキュリティテストのようにソフトウェアが動作できる状態になる前でも実施できるので潜在するセキュリティリスクを上流工程で見つけ出すことができる。このことによって、より効果的にリスクを制御することが可能になる。

脅威モデリングはいつ行うか

脅威モデリングは、ソフトウェアの設計段階の比較的初期からに行うことができる。より具体的には、設計の概要がある程度落ち着いて、次の事項の見通しがはっきりしてきた時点で行うことができる。

- そのソフトウェアシステム全体はどのような構成要素を組み合わせて実現するか
- 外部との入出力はどのようなものがあるか
- データはどのように蓄積されるか
- ソフトウェアシステム内部をおおむねどのような形でデータが流れるか

更に以降システム構成が変わった際に見直すとより有効に活用することができる。

3.1. 脅威の分類：STRIDE[27]

STRIDE はソフトウェアの脅威を識別するためのものとして次に示す 6 種の事象が起きた場合を想定してみることで実際の脅威を識別しやすくしている。

S: なりすまし(Spoofing Identity)

攻撃者が他のユーザを装ったり、悪意のあるサーバが有効なサーバを装ったりすること。

T: 改ざん(Tampering with data)

悪意を持ってデータを変更すること

R: 否認(Repudiation)

ユーザが行った操作の事実を否定すること。

否認不能性の実現は元々それなりに手間ではある。ユーザのなりすましや履歴等の記録データの改ざんができないことが前提になる。

I: 情報の漏洩(Information Disclosure)

不本意な相手に情報が開示されてしまうこと。

D: サービス妨害(Denial of Service)

正当なユーザに対してサービスの提供ができなくなること。

利用数やデータ量の制限なしにレスポンスも含めたサービス低下に対して対応することはできない。

E: 権限昇格(Elevation of Privilege)

権限を与えられていないユーザが、高い権限を取得すること。

JNSA セキュアシステム開発ガイドライン「Web システム セキュリティ要求仕様（RFP)」編 β 版[28] でも 3 案の一つとしてこの STRIDE を用いている。

---

[27] Threat Modeling, Frank Swiderski and Window Snyder, Microsoft Press, June 2004, ISBN 0-7356-1991-3.
『脅威モデル ― セキュアなアプリケーション構築』,日経 BP 出版センター

[28] http://www.jnsa.org/active/houkoku/web_system.pdf

3.2. 脅威モデリングの手順

脅威モデリングを例えば次の手順で行う。

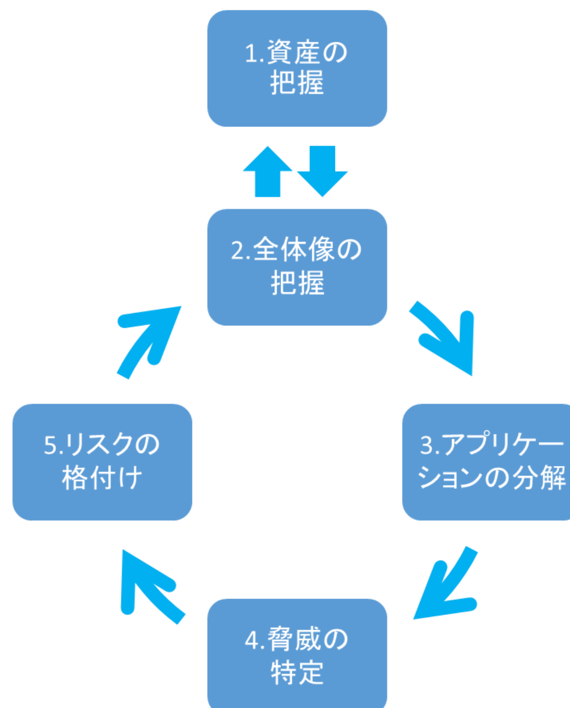

図 1：脅威モデリングの手順

1) 資産の把握

システムが守るべき資産を確認する。

2) 全体像の把握

開発するアプリケーションの主要な機能、特徴などを認識するために、全体像を確認する。 最初に、アプリケーションの構成物と構造、サブシステム、および配置の特徴を大まかに描き、セキュリティや通信メカニズムがわかるように詳細化する。

3) アプリケーションの分解

アプリケーションのアーキテクチャを構成要素に分解して解析する。信頼の境界、データフロー、入口、および出口を特定する。アプリケーションのメカニズムをよく理解していれば、脅威を明らかにし、脆弱性を検出するのは簡単になる。

4) 脅威の特定

攻撃者の動機や、対象アプリケーションのアーキテクチャと潜在的脆弱性から、そのアプリケーションに影響を与える可能性のある脅威を特定する。

潜在的な損害を考えるために本当に困ることがないかを確認する

本当に困らないならばはじめから脅威として取り扱う必要性が無くなる。

　たとえば、誰にでも公開している情報への参照で　その参照したかどうかを追跡する必要が無ければ、仮になりすましをされても特に問題にはならないはずである。

次の各項目が実現したら本当に困るならば、このソフトウェアシステムに対する脅威として取り上げ「脅威リスト」に記載する。

- 攻撃者がこの項目（ユーザやサーバ）になりすませる
- 攻撃者がこのデータを改ざんできる
- 攻撃者がこの操作を否認できる
- 攻撃者がこの情報を読み取れる
- 攻撃者がこのプロセスやデータフローへのサービスを妨害できる
- 攻撃者がこのプロセスを攻撃して権限を昇格できる

取り上げたそれぞれの脅威に対して　対象、攻撃の手法、対策などの項目を文書化する。

5) リスクの格付け

攻撃される可能性とそれによる被害規模を考慮して、より大きなリスクを与える脅威に対して高い優先度を与える。脅威によってもたらされるリスクと、それを緩和するために必要なコストを比較した結果、何の対策もとらないという判断が下されることもあり得る。ランク付けの方法の例として、上記で洗い出した脅威のひとつひとつに関し、次の尺度のそれぞれにおおまかな評価（大、中、小）を下し、この総合評価をその脅威のランクとして割り当てる方法がある。

- 「被害程度」侵害により生じる被害の程度は深刻か
- 「再現性」　脆弱性が利用できる頻度は多いか
- 「容易さ」　この侵害の実行は容易か
- 「被害者数」被害を受ける人は多いか

ただし、「再現性」と「容易さ」を考えるためには、攻撃手順が分らないと判断しきれない。これらの代わりに可能性を考えると言う意味では次の問いを考えてみる方法もある。

- 攻撃者はこの項目（ユーザやサーバ）になりすましやすいか
- 攻撃者はこのデータを改ざんできるか
- 攻撃者はこの操作を否認できるか
- 攻撃者はこの情報を読み取れるか
- 攻撃者はこのプロセスやデータフローへのサービスを拒否できるか
- 攻撃者はこのプロセスを攻撃して権限を昇格できるか

それでも判断できない場合は事前に方針として、中または大を選択することを決めておくことをした方が良い。以前はより安全側になるように常に大を選択することを推奨していたが、まずは明確に評価が高いものを優先するべきという考えから大だけでなく中を選択できることにした。

4. 開発プロセス

本章では、上流工程からの作り込みと更にその作り込んだものを確認するレビューやテストなどを取り扱う。

セキュリティはシステムテストの一部で確認することが中心という後からの対策として行われがちであった。品質と同様に開発の始めから作り込むものと意識を変える必要がある。

後からではなく、始めから取り組みが必要とされている。-本講座では、2002 年の最初の公開からこのことを訴え続けており、2010 年 8 月公開の「早めのチェック 3 工程によるセキュリティ品質確保」[29]では開発側だけでなく品質管理側からも伝えようとしたが、結果的に伝わりきれていない状態が続いている。最近では NISC でも”security by design” というスローガンを示して、設計など上流からセキュリティに取り組むことを推奨している。

また、他の要件と同様にセキュリティも最高ではなく、適切なセキュリティが必要とされている。セキュリティは非機能要件として非機能要求グレード[30]の 6 大項目のひとつとして取り扱われている。非機能要件は機能要件と比較して明確な要件が出にくい傾向がある。そのような場合に必要以上に高いレベルになったり、逆にまったく行われなかったりすることがおきている。まったく行わないは論外にしてもやりすぎても結果的に良いことはない。やりすぎていたために本来の機能に制限が起きてしまい、後でセキュリティ機能のほとんど無効化してしまうことになりかねない。すなわち適切に行うことを意識する必要がある。適切に行うためには実施することを明確にするために実施しないことを事前に明確に合意しておくことが必要になる。開発者側は使用する言語やプラットフォームが保有している既知の脆弱性を事前に認識していることが前提になり、この既知の脆弱性ことを含めて発注者側と共通の認識を持っておくことが必要である。

ソフトウェアのセキュリティ品質を確保するための次の 3 つの活動がある。

・設計工程における「セキュリティレビュー」:
　設計ドキュメントをレビューし、考慮すべきセキュリティ対策に漏れがないか検証する
・実装工程における「ソースコードレビュー」:
　記述したソースコードを直接検証する
・テスト工程における「セキュリティテスト」:
　作り上げたプログラムを動作させて、テストし、セキュリティ脆弱性を見つけ出す

[29] https://www.ipa.go.jp/security/awareness/vendor/programmingv2/contents2/
[30] https://www.ipa.go.jp/sec/softwareengineering/reports/20100416.html

4.1. セキュリティレビュー

⑴ 目的

セキュリティレビューは、セキュリティ対策漏れを早くに見つけ出し、設計者へフィードバックすることを目的に行う。

次のような観点で確認する。

・セキュリティポリシーを満たしているか

・セキュリティ要件に対する対策漏れがないか

・既知の脆弱性対策が行えているか

⑵ 対象

セキュリティレビューを行う際のレビュー対象は、設計工程までに作成される　セキュリティ要件の定義書、ソフトウェア構造、業務仕様、モジュール分割の設計書、テストの計画書などである。

4.2. ソースコードレビュー

昔はコンピュータリソースの制約からソースコードレビューを普通に先に行われていたが、現状の即時にコンパイル実行ができる環境ではあまり重要視されなくなってきている。ただし、昔のソースコードレビューでセキュリティが取り扱われことはほとんどなかったとは考える。

ソースコードレビューは、開発者担当者が書いたソースコードを閲読して読みやすく、分り易く書かれているか、更にはセキュリティ脆弱性あるいはそのきざしがないか読み取る作業である。

ソースコードレビューで見いだされた脆弱性の情報を開発作業にフィードバックし、ソースコード修正を行う。

⑴ ソースコードレビューを行う理由

プログラマが書き下ろしたソースコードは、そのままでは多くのバグといくつかのセキュリティ脆弱性を内に秘めているおそれがある。多くの担当者が関わる現状からすると後の保守や機能改善等に向けて、ソースコード自体がプロジェクトの標準にしたがって一定の品質であることは非常に重要な要素のひとつであるため。

(2) 誰が行うか

ソースコードレビューを行う人物には次が考えられる。

・開発者本人
・開発チームの別の担当者
・開発チームとは独立したセキュリティ脆弱性対策チームのメンバ

開発者本人がソースコードを見直すことと、別の人物の眼によるチェックを行うことの両方が行われるのが望ましい。

前者の利点は、開発者がより脆弱性の少ないコードを書く能力の向上がはかれること、後者の利点は、思い込みによって開発者が見落としがちな問題についても見つけ出す機会が得られることである。

レビューという独立したタスクとその記録を形式知として残すことには適さないが、ペアプログラミングというやり方を用いて常に複数人の目を通して改善を進めるというやり方もある。

(3) いつ行うか

ソースコードが新たに書きおこされるか、既存のプログラムが改修されて、ソフトウェアの中のひとまとまりの機能のソースコードが揃った時点でレビューを行う。

セキュリティのソースコードレビューの前また同時に品質としてソースコードレビューを実施しておくと良い。

(4) ソースコードレビューの実施

脆弱性検出のためには、少なくとも次のような観点を意識する。

・プロジェクトで定められたセキュアコーディング標準にしたがっているか？
・既知の脆弱性対策を行えているか？
・セキュリティ上注意を要するライブラリ関数を呼び出している箇所はないか？
・設計ではもともと予定されていないデバッグ機能や保守機能等がプログラマの判断で組み入れられていないか？情報漏えいやアクセス制御の迂回がおこらないか？
・領域からデータがあふれ得る箇所はないか？

ツールの利用

より手軽に始める方法としては、最初はソースコードの静的検査ツールを使い、ツールが自動で検出してくれる問題から手を付けるという方法がある。更に CI(継続的統合)ツールと組みまわせて、常にシステム構成管理にチェックインされた際に自動的にツールの範囲でのチェックを実施することも行われるようになりつつある。

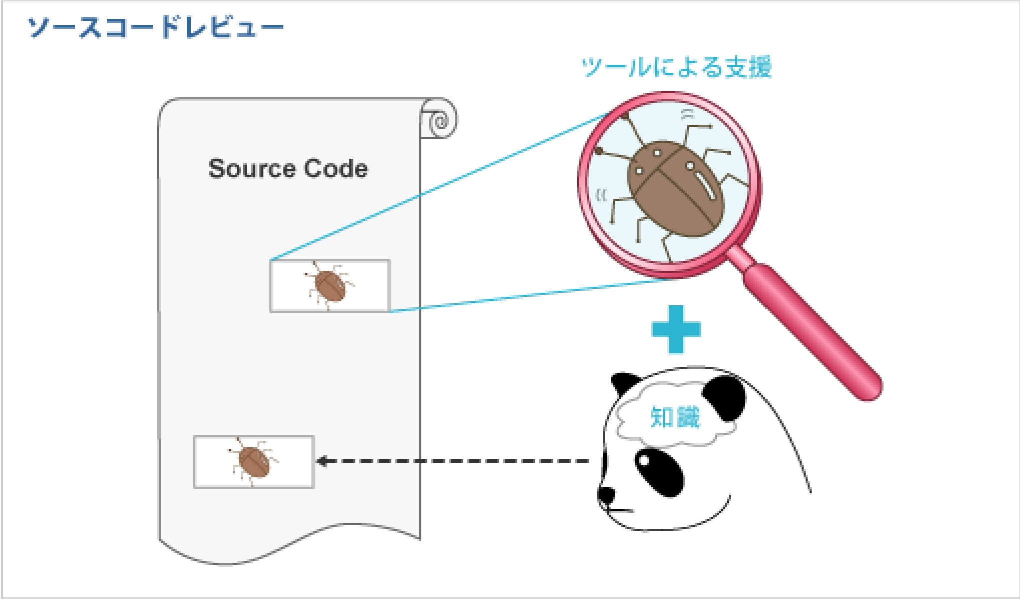

図 2:ソースコードレビュー

4.3. セキュリティテスト

(1) 目的

セキュリティテストの目的は、作り上げたプログラムに十分なセキュリティ対策が実装されているかどうかを確認することである。

次のような観点で確認する。

・既知の脆弱性パターンに陥っていないか
・セキュリティ設計通りの実装ができているか
・設計工程までに検討しきれなかった脆弱性がないか

(2) 仕様・要領

セキュリティテストで確認するのは、セキュリティ要件にあがり、設計に反映されている項目は、当然、実装も行われているはずなので、実装漏れが無いか、セキュリティテストでも確認する必要がある。

また、ソフトウェアのテストの工程が、

単体テスト → 結合テスト → システムテスト

と進む中で、どのテスト工程においても必要十分なテストを行う必要がある。

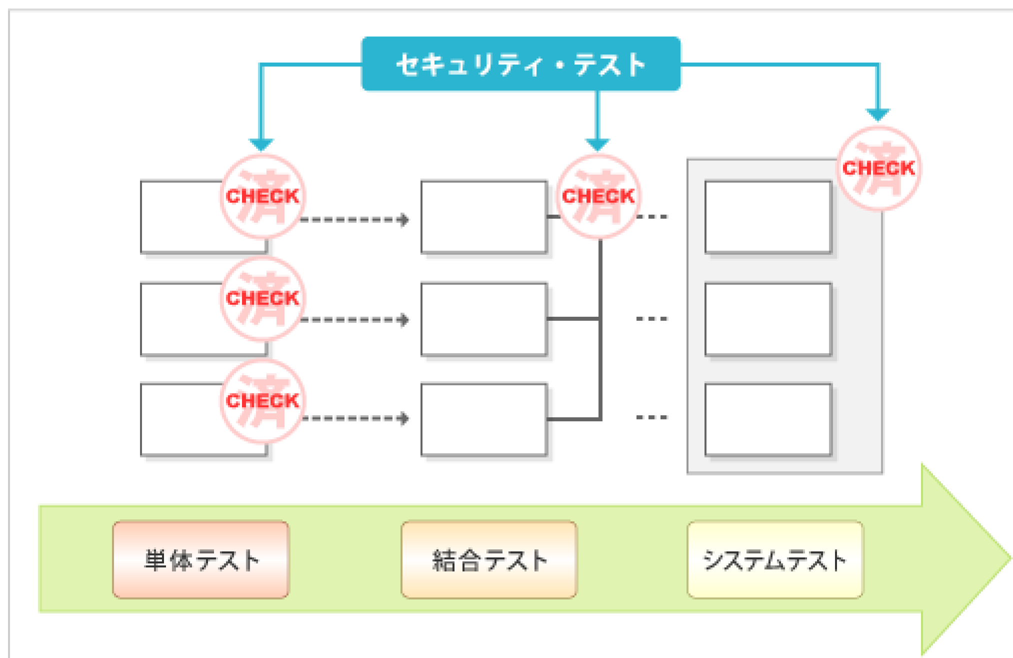

図 3: セキュリティテスト

(3) 手段

セキュリティテストは、通常のテストと同様に次の 2 種類のやり方がある。

1) ブラックボックステスト

一般的にブラックボックステストとは、プログラムの内部の作りとは関係無く、外部からシステムの機能をテストする方法である。

セキュリティテストで行うブラックボックステストでも、同様に外部からの入力によるプログラムの動作を確認する。 また、このテストではツールを使って一定の問題を検出する方法が利用できる。 ツールによる検査では、熟練した脆弱性対策技術者でなくてもテストが行える利点があるが、結果を正しく判断するにはそれなりのスキルが必要になる。

2) ホワイトボックステスト

一般的にホワイトボックステストとは、プログラムの内部の作りを理解した上で予定された処理が、予定通り動作するかどうかを確認するテストである。

セキュリティテストで行うホワイトボックステストでも、同様に処理の条件や組み合わせなどによるテストを行う。プログラム仕様にあまり左右されることがなく、比較的単純な実装時だけで対応できるのであれば、ブラックボックステストと同様に熟練した脆弱性対策技術者でなくてもテストが行える可能性はあるが、そうでない場合には脆弱性の知識に加えてプログラムの仕様とプログラミング言語の知識も相応に必要となるため現実的には非常に難しいと考える。

(4) 注意事項

セキュリティテストを行うにあたり、注意すべき項目を次にあげる。

- どのようなテストを実施するかは、設計段階で決定しておく必要がある。
- 同様の項目があることを理由にしてテストを後の結合テストやシステムテストに後送りしてはならない。間違っても単体テストの代替を結合テストでとか、結合テストの代替をシステムテストで行ってはならない。
- テストパターンを想定する場合、複数の条件、モジュール、パラメータなど多くの要素を全て漏れなくテストするようにすると膨大な量になってしまう。要点を絞る必要がある。

ここまで 4.1 から 4.3 でセキュリティレビュー、ソースコードレビュー、セキュリティテストに関して記述したが、これらの確認作業はその確認すべき元の実作業が確実に実施されていることを確認するためのものであるので、くれぐれも元の実作業自身をおろそかにしてはならない。

5. 計画すべきセキュリティ機能

本章では、セキュアなシステムを構築する際に一般的に必要になるセキュリティ機能を取り扱う。2.1 設計原則でのセキュリティメカニズムや防御メカニズムの一部を担うものである。本書の「はじめに」で取り上げた　リスク＝【情報資産】×【脅威】×【脆弱性】の脅威部分に対しての対策として作り込まれる。

アプリケーション毎に必要とするセキュリティ機能は異なるが、共通して必要とされる機能はある。これらの機能はアプリケーション毎に必ずしも開発されるべきものではなく、更にいくつかは複数のアプリケーションで共有・共用したほうが良い場合もある。

ここでは次の 7 つを取り上げる。
認証、認可、暗号化、入力の検証、出力のエスケープ、例外処理、ロギング
一般的にこれらの機能を用いないでセキュアなシステムを構築することはない。システムのアーキテクチャ検討時からこれらに関しても取り入れられるべきものである。

マイクロソフトの「Web アプリケーションの脅威モデル[31]」でのセキュリティフレームでは、上記 7 つのほかに構成管理、機密データ、セッション管理、パラメータ操作の 4 つがある。

機能の多くの部分は JPCERT/CC で日本語版も公開している「OWASP アプリケーションセキュリティ検証標準（以降　OWASP ASVS と記す）[32]」を参考した。機能の要件等の詳細に関しては該当する OWSP ASVS の章を各機能の最後に明記したので参照していただきたい。但し OWASP ASVS は、Web アプリケーションを対象として記述されているため、Web 以外のアプリケーションでは該当しない場合もある。

---

[31] Web アプリケーションの脅威モデル
https://msdn.microsoft.com/ja-jp/library/ff648006.aspx
https://msdn.microsoft.com/en-us/library/ff648006.aspx
[32] https://www.jpcert.or.jp/securecoding/OWASP_ASVS_20160623.pdf

5.1. 認証

認証とは、アクセスしてきている対象が本物であることを確認する手続きである。　対象が人の場合には、本人であることを確認するものである。

認証には複数の方式があるが、どの方式の場合も次のふたつのものを対象から受け取って判定を行う。

- 対象を一意に識別する識別情報
- 対象しか提示できない秘密情報

ユーザは、誰か。認証とは、対象が本物であること、つまり、対象についてなされた主張が真実であることを立証または確認する行為である。
確認に用いる秘密情報は安全に転送される必要がある。また、秘密情報はひとつとは限らず、異なる種類の複数要素を確認する多要素認証が用いられることが多くなっている。更に認証をアプリケーション個別に行わず、複数のアプリケーションで連携して扱うことも多くなってきている。
参考：OWASP ASVS の V2: 認証に関する検査要件

5.2. 認可

認可とは、リソースへのアクセスを、当該リソースの使用を許可されている対象にのみに制限するものである。対象のアプリケーションが次の要件を満たすこと。

- リソースにアクセスする対象が有効な認証識別情報を保持していること
- リソースにアクセスする対象に対する権限のセットと認証識別情報が関連付けられていること

ユーザは、何ができるか。 認可とは、アプリケーションがリソース、および操作へのアクセス制御を提供する手法である。
参考：OWASP ASVS の V4: アクセス制御に関する検査要件

5.3. 暗号化

暗号化機能は、理論的にも安全性が確認されているものを用いる。昔は独自の暗号化機能を作り込むことが行われたが、それらの多くは必ずしも安全性が確認されたものではなかった。鍵の更新は、必要に応じて実施できるように運用も合わせて考慮しておく必要がある。さらに長期的に運用されるシステムでは、暗号方式の変更される可能性も考慮しておく必要がある。
秘密（機密）をどのように保持するか、データの非整合、ライブラリの誤った使用をどのように防止するか、暗号化法として強度を左右する乱数の元をどのように提供するかなど暗号化では、アプリケーションが機密性および整合性をどのように保持するかに注意する。パスワードのように可逆性が必ずしも必要でない場合には、ハッシュを用いて、更にソルトとストレッチングを行うようにする。
対象のアプリケーションが次の要件を満たすこと。

- 全ての暗号化モジュールにおいて処理の失敗がセキュアに実施されており、エラーが適切に処理されること
- ランダム性が必要な場合は適切な乱数生成器が使用されていること
- 鍵へのアクセスが安全な方法で管理されていること

参考：OWASP ASVS の V7: 暗号化に関する検査要件

5.4. 入力の検証

入力の検証（バリデーション）は、開発者が想定したものだけを認めるようにする。検証する際には、事前に標準化や正規化も行う必要がある。アプリケーションが受け取る入力が有効で、かつ安全であることをどのように確認するか、アプリケーションが他の処理を実行する前に入力をフィルタ、変換、または拒否する方法に注意する必要がある。
Web アプリケーションセキュリティの最も一般的な脆弱性は、クライアントまたは環境から受信した入力が検証されないまま使用されることにある。この脆弱性は、インジェクション系の攻撃、ロケール / Unicode 攻撃、ファイルシステム攻撃、バッファオーバーフローなど、Web アプリケーションのほとんどの主な脆弱性に関係している。
但し、この機能は根本的解決策ではなく補助的な保険的対策となる場合が多い。
対象のアプリケーションが次の要件を満たすこと。

- すべての入力が正しく、本来の目的に適合していること
- 外部のエンティティまたはクライアントからのデータは、信頼できないものとして扱われるべきである。

参考：OWASP ASVS の V5: 悪性入力の処理に関する検査要件

5.5. 出力のエスケープ（無害化）

出力先に合わせて適切にエスケープする必要がある。
どこに出力しているかを明確に意識し、そこでは何をエスケープされないといけないかを理解しておくことが必要になる。
既存のエスケープ機能を使う場合には、意図したように本当に機能するかを事前に確認しておく必要がある。
・SQL を扱う DBMS ならば SQL インジェクション対策が行われていること
・HTML の BODY ならば XSS 対策が行われていること
・HTML の HEAD ならば HTTP ヘッダ・インジェクション対策として改行コードを排除していること
参考：OWASP ASVS の V6: 出力のエンコード / エスケープ

5.6. 例外処理

例外処理が占める割合は、正常系の何倍にもなることが、更に例外処理は必ずしも正常系と同様に確認しきれないことが多いので脆弱性が残ったままになってしまう場合がある。
開発中や不具合の調査の際には、できるだけ多くの情報が開発者に必要になるが、その情報の内一般の利用者に伝えるべき情報は決して多くはない。
アプリケーションでメソッド呼び出しが失敗した場合、次のことがどうなるか事前に確認しておくことが必要になる。
アプリケーションは何をするか。 どの程度、例外を明らかにするか。 エンド ユーザにわかりやすいエラー情報を戻すか。有用な例外情報を呼び出し元に戻すか。アプリケーションは、適切に失敗となるか。

エラー処理の主な目標は、ユーザ、管理者、およびインシデント対応チームを適切な形で支援することである。
参考：OWASP ASVS の V8: エラー処理とログの保存に関する検査要件

5.7. ロギング

ログは、ユーザ、管理者、開発者、およびインシデント対応チーム等を適切に支援できるように保存されていなければならない。単に大量なだけでなく、有用なデータを漏れなく確実に正確に保存されており、書き換えられていないことが必要である。
監査や否認防止という役割をもつこともあり、ログ自体が改ざんされていないことを確認できることが必要になる場合もある。
誰か、いつ、何を行ったか。ログ記録では、アプリケーションがセキュリティ関連イベントを記録する手法に注意する必要がある。
ログ自体の改ざんや喪失を防ぐために別のサーバ上に保存し、システムが侵害されてもログデータを保全できる仕組みが必要な場合もある。
例えば通信ログとして単純にすべての情報を保存すると機密データが含まれてしまう可能性がある。だからといって一部しか保存しないと、監査や否認防止には役立たないことにもなりかねない。

ログには、機密のデータが含まれる場合があるため、データプライバシー法令等に従って保護する必要がある。

- 特別必要な場合を除いて、機密情報の収集やログの保存は行わない
  例えば、GETパラメータでID、PASSWORDを扱うと通常WEBサーバのログに保存され露呈されてしまうことになる場合がある。
- ログに保存された情報は、セキュアな方法で処理し、保護する
- ログは永続的なものとしてはならない

ログにプライベートまたは機密データが含まれる場合、ログは攻撃者にとってきわめて魅力的なものとなる。
参考：OWASP ASVS の V8: エラー処理とログの保存に関する検査要件

おわりに

本書では、セキュアなシステムを構築するためセキュア・プログラミングを実現するためにするべき原則を中心に記述した。これまでの「セキュア・プログラミング講座」と異なり、セキュア・プログラミングにおいて共通するものを記述した。
従来記載されていた記述のうち、個別のものに関しては、後述の付録に記述したので参照していただきたい。

最後に、本書がアプリケーションの開発に活用され、セキュリティ強化の一助になることを期待している。

付録

過去のコンテンツに関して今後更新を行わないのでその補足情報を付録として提供する。但し、2002 年 2 月に公開開始したセキュプログラミング講座(旧版) の言語編の Perl や VBScript/ASP やプラットフォーム編に該当する部分はない。

付録 A:Web アプリケーションに関して

Web アプリケーションはプロトコル HTTP を用いるアプリケーションである。

通常ブラウザと呼ばれるクライアントプログラムからサーバにアクセスして使用する。

元々は次に示すような非常に単純な構成だったと言える。

1． クライアント側のブラウザ上では画面からユーザ入力を受付、リクエストとしてサーバに送る。
2． サーバ側ではそのリクエストを入力として受取、処理して、レスポンスとして出力をクライアントに返す。
3． クライアントのブラウザではサーバからのレスポンスを受け取り、それをブラウザ上でレンダリングして表示する。

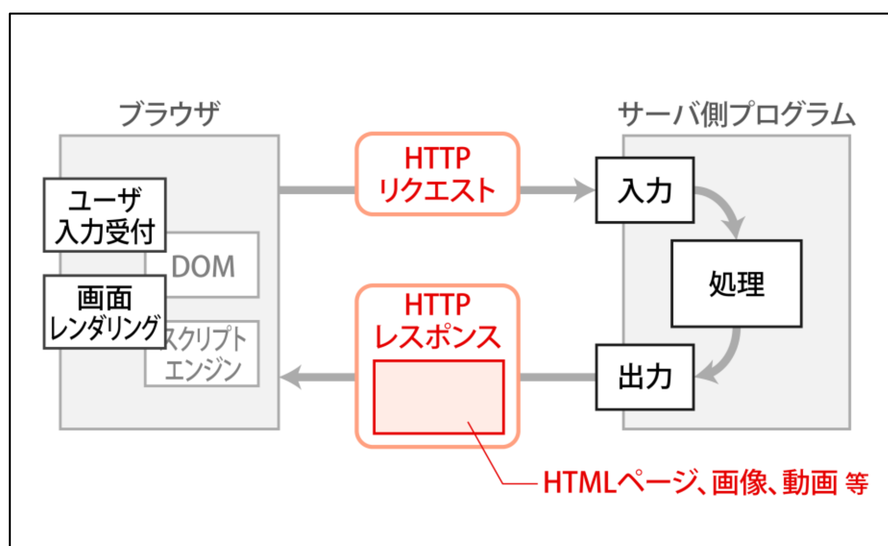

図 4：Web アプリケーションの概略構造

最も単純な使い方としては、ブラウザから静的な HTML ファイルのリクエストをサーバに送り、サーバがその HTML ファイルの内容をレスポンスとして返してくるというきわめて単純なステートレスな構造であった。サーバ上に特に何の状態や情報も保持する必要がない構造のために、非力なマシンでも容易にかなりの多くの不特定多数に対してサービス提供が可能となった。

その後この仕組みを使ってアプリケーションを動かすことが考えられ、サーバ側で Java、PHP、C#、Perl、Ruby、Python など多くの言語が使用したアプリケーションが作られるようになった。この際にサーバ側に認証情報を始めとしてその後やり取りでの何だか情報を保持することが望まれ、セッション管理というステートフルな仕組みが取り入れられた。始めは各々のアプリケーション自体でこの機能を作り込みが行われ、結果的に必ずしも十分に機能仕切れなかったため脆弱性の原因になった。最近ではフレームワーク

等でこのセッション管理機能が提供されるので提供された機能の使い方を間違わなければ問題が起こらなくなりつつある。

『セキュア・プログラミング講座(Web アプリケーション編)』ブートアップセミナー資料[33]にも紹介されているので参照されたい。

HTTP プロトコル上は可変長の文字データをクライアントからサーバにリクエストとして送り、その返答をレスポンスとして受け取ると極めて単純な構造は維持されている。このため、必ずしも既製のブラウザで無くても勝手にリクエスト文字列を生成してサーバに送ることも可能になってしまっている。

代表的な脆弱性に関しては「安全なウェブサイトの作り方[34]」参照されたい。
具体的なセキュアコーディングに関しては JPCERT/CC のラーニングのセキュアコーディング[35]の次の資料を参照されたい。

·Java セキュアコーディングセミナー資料
·Java セキュアコーディングスタンダード CERT/Oracle 版
·書籍：Java セキュアコーディングスタンダード CERT/Oracle 版

33 https://www.ipa.go.jp/files/000030878.pdf
34 https://www.ipa.go.jp/security/vuln/websecurity.html
35 https://www.jpcert.or.jp/securecoding/index.html

付録 B:C/C++言語に関して

C 言語は、1972 年に AT&T ベル研究所で開発した言語である。アセンブラとの親和性が高く UNIX をはじめとして OS の記述言語としても用いられている。C 言語の精神としてプログラマを信頼し、プログラマがやりたいことができるようにするためのものとして作られている。言語仕様には未規定、未定義、実装定義事項が存在している。そのためコンパイラ毎の非互換性があり、同じソースコードでも動作が同じになるとは限らない。
言語仕様の詳細を知らないプログラマがこの違いを十分に理解せずに作成したプログラムは、異なる環境では元とは異なる動きとなり、脆弱性を作り込むことが起きている。

セキュアコーディングに関しては JPCERT/CC のラーニングのセキュアコーディング[36]の次の資料を参照されたい。

・C/C++ セキュアコーディングセミナー資料[37]
・CERT C セキュアコーディングスタンダード[38]
・書籍 :「CERT C セキュアコーディングスタンダード」及び「C/C++ セキュアコーディング 第 2 版」

更に組込みソフトウェア開発向けならば IPA　SEC から提供されている次のものを参照されたい。

・【改訂版】組込みソフトウェア開発向けコーディング作法ガイド［C 言語版］Ver.2.0[39]
・組込みソフトウェア開発向けコーディング作法ガイド［C++言語版］[40]

36 https://www.jpcert.or.jp/securecoding/index.html
37 https://www.jpcert.or.jp/research/materials.html
38 https://www.jpcert.or.jp/sc-rules/
39 https://www.ipa.go.jp/sec/reports/20140724.html
40 https://www.ipa.go.jp/sec/softwareengineering/reports/20130329_2.html

「セキュア・プログラミング講座」

［発　　　行］　第一版 2016 年 10 月 31 日
［著作・制作］独立行政法人情報処理推進機構　技術本部　セキュリティセンター
［執　筆　者］塩田　英二