平成30年度「専修学校による地域産業中核的人材養成事業」

セキュアなネットワーク設計

Society5.0に対応した情報セキュリティ人材養成のモデルカリキュラム開発・実証事業

平成 30 年度「専修学校による地域産業中核的人材養成事業」

セキュアなネットワーク設計

Society5.0 に対応した情報セキュリティ人材養成のモデルカリキュラム開発・実証事業

目　次

第１章

ネットワークシステムのリスクと脆弱性

第１章　ネットワークシステムのリスクと脆弱性

1. セキュアなネットワークシステム構築の基本的な考え方

1-1. 基本的な考え方

本章では、セキュアなネットワーク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行うべき原則を中心に記述する。内容は、ソフトウェア製品やWebアプリケーション等において共通するものとなる。

セキュアなネットワークシステムを構築するためには、要件定義、設計、コーディングの段階で脆弱性を作り込まないようにする考え方が必要だ。

「脆弱性」とは、「ソフトウェア等におけるセキュリティ上の弱点」のことで「セキュリティホール」とも呼ばれている。開発時に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のことであり、「脆弱性」≒「攻略可能性」とあらわすことができる。システムエンジニアやプログラマが習得する通常のプログラミング・テクニックの知識だけでは、それと知らずに「脆弱性」を発生させてしまうことがある。

近年、脆弱性がコンピュータウイルスや不正アクセス等の攻撃に悪用されるケースが増加している。また、脆弱性に関する情報の公開後に、その脆弱性を狙う攻撃方法が作られ、広まるまでの期間が短くなる傾向があり、公開された脆弱性の対策前にコンピュータウイルスに感染する危険性や公開サーバが攻撃され大きな被害を受ける危険性、および脆弱性を放置したことにより第三者が被害を受ける危険性も増大している。

これら危険性の増大に伴い、ソフトウェア開発のライフサイクルプロセスの枠組みにおいても、脆弱性の対処する段階は変わってきている。

従前は、実装後のテスト・運用の下流工程で脆弱性確認を行い、対処されていた。下流工程で短期に修正することが可能な脆弱性も相当数にのぼるが、実際には根本的にやり直さなければならないプロジェクトが多い。そのため、実装後に脆弱性を解消するのは「容易ではない」、「手戻りによる修正のコストや納期への影響が大きい」と考えられるようになってきた。

今般のセキュア・プログラミング講座などでは、要件定義や設計段階の上流工程から問題を認識することの重要性が説かれている。システムのセキュリティ被害は、攻撃を前提とした設計を行うことや、コード自体に含まれる脆弱性を減らすことで、その発生を抑えることができる。どの工程から対策に着手するのが相応しいかは脆弱性の種類によって異なるものの、事前に「脆弱性」を考慮しておくことが重要となる。

図解 1-1　ソフトウェア開発のライフサイクルプロセス

出所：独立行政法人情報処理推進機構（IPA）「セキュア・プログラミング講座のねらい」

1-2. 情報セキュリティの概念

情報セキュリティとは、保護すべき情報や情報システムを安全な状態にすることをいう。JISQ27002（ISO/IEC 27002）では、情報セキュリティについて「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、セキュアであるためには、機密性（confidentiality）、完全生（integrity）、可用性（availability）の 3 つが維持されており、さらに真正性（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必要がある。なお下記 3 要素について、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。

① 機密性（confidentiality）：

ユーザーが許可されているデータへのアクセスのみを許可する。

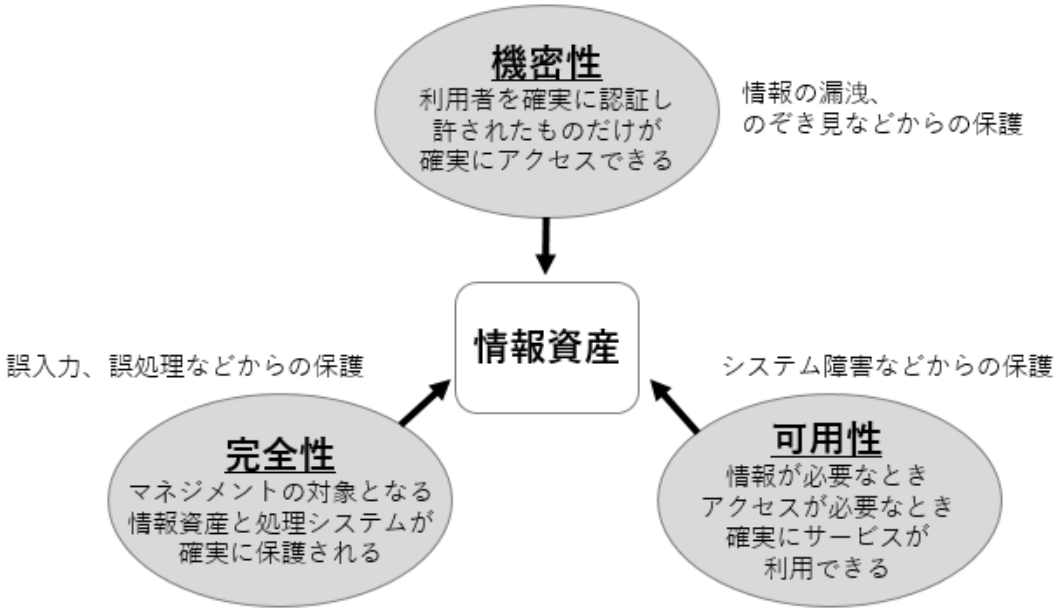
② 完全性（integrity）：

許可されていないユーザーによってシステムが変更されたり改ざんされたりしないようにする。

③ 可用性（availability）：

許可されたユーザーが必要なときにシステムとデータを利用できるようにする。

図解 1-2　情報セキュリティの定義

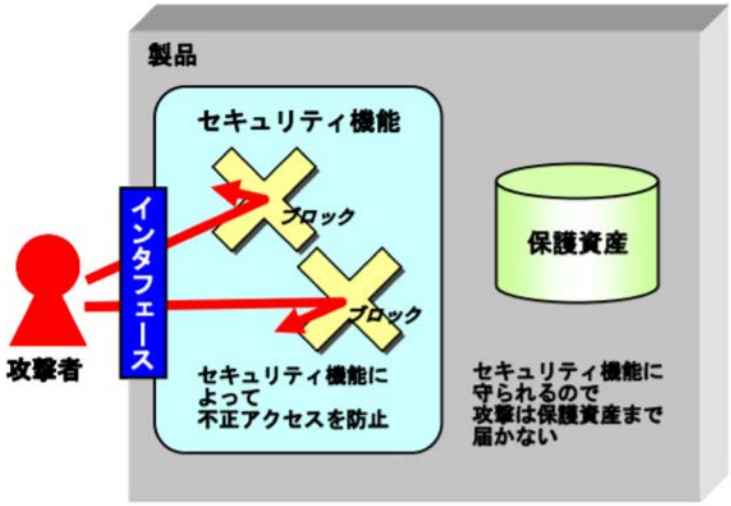


出所：UNISYS TECHNOLOGY REVIEW 第 82 号
「企業統治に欠かせない情報セキュリティの実装」

1-3. 情報セキュリティリスク

情報セキュリティリスクは情報セキュリティを損ねる要因となるものであり、「システムが想定した通りに機能しない可能性、悪意ある外部者または偶発的な要素による妨害・加害行為、内部者の故意・ミス・不作為による事故」(JNSA[1]) がある。これら情報セキュリティリスクに対し、情報（保護資産）がセキュリティ機能によって守られている概念を図解 1-3 で示す。

図解 1-3　セキュリティ機能による保護



出所：独立行政法人情報処理推進機構（IPA）
「開発者のためのセキュリティ解説書」(2018 年)

情報漏洩や不正アクセス、コンピュータウイルスの感染など、主に企業のコンピュータに関する情報管理やシステム運用に関するセキュリティ上の脅威となる出来事をセキュリティインシデントという。典型的なインシデントとしてあげられるのは、マルウェア感染やネットワークからの不正侵入によるシステムの停止、データの破損、情報の窃盗、人的な不注意やミス、故意による情報の紛失や漏洩、持ち出し、悪用等である。

[1] 日本ネットワークセキュリティ協会　https://www.jnsa.org/

多くの場合、インシデントは企業の事業等の健全な運営に悪影響を与えたり、ネットワークセキュリティに甚大な被害をもたらしたりする。銀行口座の情報、技術情報などの重要情報が第三者に悪用されることによる経済的被害や技術競争力の損失、またこれらの事故の二次被害としての風評被害、損害賠償負担、信用の失墜、機会損失等、インシデントによる被害例は多岐にわたる。

ネットワーク上のデータが漏洩しても問題のない情報だけであれば、脅威も脆弱性も考える必要はない。また、問題等が生じる脅威がなければ、脆弱性があったとしても被害等にはつながらない。つまり、リスクについては以下のように考えることができる。

リスク（損害が発生する可能性）＝【情報資産】×【脅威】×【脆弱性】

この考え方では、リスクをなくすことは情報資産、脅威、脆弱性のいずれかをなくすことで可能となり、損害を防ぐことができる。

より堅固なネットワークシステム構築のために、脅威と脆弱性に対しどのような対策をとり、どの段階で対策をとるべきか。Web プログラマ、製品プログラマ、Web アプリケーション、C/C++言語に共通する原則を中心に記載していく。

2. 脆弱性対応の基本

2-1. 脆弱性とは何か

脆弱とは「もろくて弱い様子」を意味し、ソフトウェアを含めた IT システムにおける脆弱性（Vulnerability）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」を意味する。

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。ソフトウェア製品及び Web アプリケーションに関する脆弱性関連情報の円滑な流通と対策の普及を目指す「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。

脆弱性が残された状態でコンピュータを利用することにより、以下のような危険性がある。

・不正侵入
・マルウェア感染
・なりすまし
・情報漏洩・改ざん
・盗聴
・サービス不能

---

[2] 総務省「国民のための情報セキュリティサイト」
http://www.soumu.go.jp/main_sosiki/joho_tsusin/security/basic/risk/11.html

[3] 情報処理推進機構「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」
https://www.ipa.go.jp/security/ciadr/partnership_guide.html

2-2. 既知の脆弱性への対応

脆弱性への対策は、取り組みの視点や対策内容により、期待できる効果が異なる。脆弱性の原因そのものを取り除き、根本解決が期待できる対策がある一方、外因である攻撃手法に着目することにより特定の攻撃は防御できたものの、同じ対策が別の種類の攻撃に対しても有効とは限らない。選択する対策がどのような性質を持ち、期待する効果を得られるものなのかを正しく理解把握することが重要となる。

脆弱性の対策に関する根本的解決とは、「脆弱性を作り込まない実装」を実現する手法を実施することといえる。根本的解決を実施することで、その脆弱性を狙った攻撃を無効化することが期待できる。

すでに運用を開始している Web アプリケーションにセキュリティ上の問題が発覚した場合、設計レベルから修正することは難しい場合が少なくなく、場あたり的な対策で済まさざるをえないこともある。

近年は標的型攻撃やフィッシングが増加しており、問題発見後の対応に重点が置かれがちであるが、これらの被害に遭った場合も、OS やソフトウェアの脆弱性がなければ被害を抑えることができる。

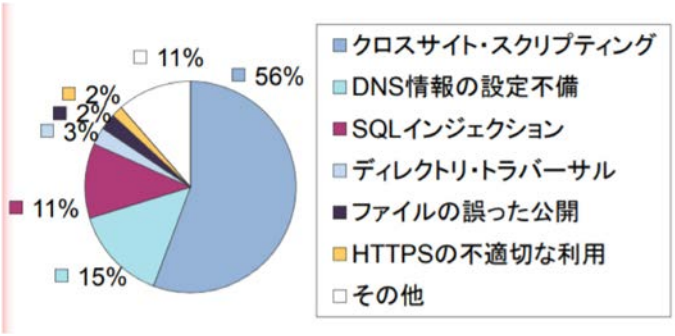
重要なのは、計画・設計から構築までに、脆弱性の検査、修正などの対策を行うことだ。特に、頻繁に出現する「作り込まれやすい」脆弱性については、設計・開発段階で未然に解消することが望ましい。

一方、開発者も、個々のアプリケーションの既知の脆弱性に関して必ずしも十分に認識できていないことがある。また、対策をしても減るどころか増える一方の攻撃に対し、いたちごっこのようになってしまう。そうした状況を改善するためにも、脆弱性対策を考慮した開発を行うことが重要となる。というのも、Web アプリケーションの代表的な脆弱性は新しいものではなく、根本的解決が不可能ではないからだ。特に、クロスサイト・スクリプティング（Web アプリケーションの脆弱性もしくはそれを利用した攻撃）や SQL インジェクション（アプリケーションのセキュリティ上の不備を意図的に利用し、アプリケーションが想定しない SQL 文を実行させることにより、データベースシステムを不正に操作する攻撃方法）等の代表的な脆弱性の仕組みは昔から存在して

おり、これらはプログラミングの際に作り込んでしまうケースが大半となっている。開発段階からこれらの対策を組み込むことで、修正等の手戻りの可能性も低減できる。

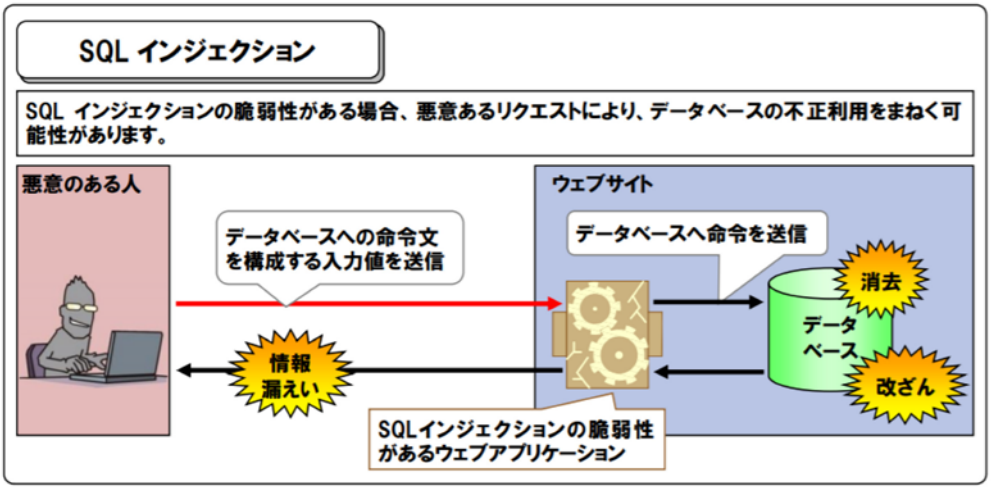
独立行政法人情報処理推進機構（IPA）では、2004 年 7 月からウェブサイトの脆弱性情報の受付を開始し、2017 年 6 月までに累計で 9,564 件の届け出があった。図解 1-4 の通り、クロスサイト・スクリプティングが過半数を占め、影響の大きい SQL インジェクション（図解 1-5）が全体の 1 割に達している。

図解 1-4　ウェブサイトの脆弱性情報の届出の累計 9326 件の内訳
（2004 年 7 月～2017 年 6 月末時点）



出所：独立行政法人情報処理推進機構（IPA）
「ウェブサイトにおける情報セキュリティ対策のポイント」（2017 年）

図解 1-5　SQL インジェクション



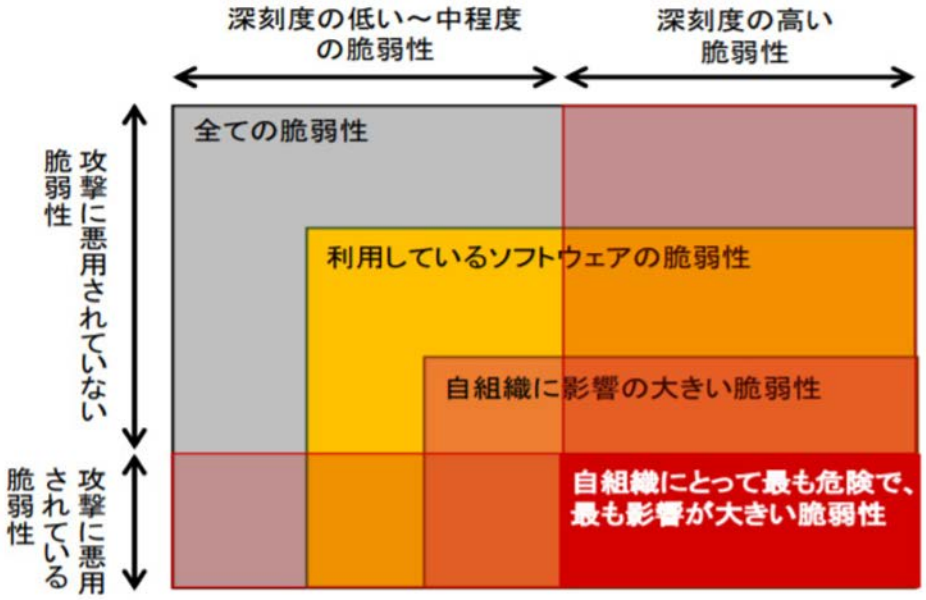
出所：独立行政法人情報処理推進機構（IPA）
「安全なウェブサイトの作り方改定第 7 版」（2015 年）

既知の脆弱性は、判明しているすべての脆弱性に対して根本的対策を行うべきと考えることができる。しかしながら、必ずしも脆弱性のすべてに根本的対策を行う必要があるわけではない。たとえば、読み込むデータが 1 万行もないことが明白であるのに、自身が作成したプログラムを今一度だけ実行する際、プログラム上の整数値のオーバーフロー問題が起きる可能性があるからといって、プログラムを本当に修正する必要があるかということがある。

また、ハードウェア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。十分なメモリがあることにより対処可能な脆弱性の場合、メモリ不足によって実際には対処できないといったこともあるのだ。

ソフトウェアが内包する脆弱性は、必ずしも危険度が高いものばかりではない。攻撃の可能性が低いもの、技術的な深刻度が低いものまでさまざまなタイプがある。やみくもに対策をするのではなく、①脆弱性の技術的特性　②攻撃情報　③影響　を整理したうえでリスクを把握し、最適な対策を行うことが重要となってくる。

図解 1-6　組織にとっての危険度を考慮した、優先的に対策すべき脆弱性の絞り込み



出所：独立行政法人情報処理推進機構（IPA）
IPA テクニカルウォッチ「脆弱性を悪用する攻撃への効果的な対策についてのレポート」

脆弱性の内容確認や対策の実施判断には、製品ソフトウェアの脆弱性を日々収録しているインターネット上のコンテンツを役立てることができる。悪用を防ぐために、詳細な実装ミスの内容や攻撃手口までは明かされない場合が多いが、リスクの把握や対策の要否を判断する参考になる。

以下に例をあげておく。

●脆弱性関連情報に関する届出状況（独立行政法人情報処理推進機構 IPA）

（https://www.ipa.go.jp/security/vuln/report/press.html）

IPA に届け出られた脆弱性の概要と統計情報を開示。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状況である。

●JVN（Japan Vulnerability Notes）

（https://jvn.jp/）

日本で使用されるソフトウェア製品などの脆弱性関連情報とその対策情報を提供し、情報セキュリティ対策に資することを目的とする脆弱性対策情報ポータルサイト。「情報セキュリティ早期警戒パートナーシップ」制度にもとづいて、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おり、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。米国 US-CERT、CERT/CC および英国 CPNI （NISCC）など海外の調整機関とも連携している。IPA 同様、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。

●CVE（Common Vulnerabilities and Exposures）

（http://cve.mitre.org/）

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。組織 A の発行する脆弱性対策情報と、組織 X の発行する脆弱性対策情報

とが同じ脆弱性に関する対策情報であることを判断したり、対策情報同士の相互参照や関連付けに利用したりできる。脆弱性検査ツールや脆弱性対策情報提供サービスの多くが CVE を利用している。

●CWE（Common Weakness Enumeration）

（http://cwe.mitre.org/）

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ている。

●SecurityFocus

（http://www.securityfocus.com/）

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。

このほか、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）」（https://www.ipa.go.jp/security/technicalwatch/20150331.html）や「脆弱性対策コンテンツリファレンス」（https://www.ipa.go.jp/files/000051352.pdf）も参考にしてほしい。

2-3. 未知の脆弱性への対応

通常、新たな脆弱性が発見されてから対策が施されるまでの流れは、脆弱性の発見→関係機関に報告→修正プログラムの作成→プログラムの配布となる。

一方、攻撃者によって脆弱性が発見されたり、修正プログラムが作成される前に脆弱性が公表されたりした場合、脆弱性があるソフトウェアは攻撃に対して無防備な状態となる。このような状態は、サイバー攻撃者にとって魅力的な標的となる。

また、攻撃者がウィルスソフトで検知されないことを確認した未知のマルウェアを作ることもある。

このような未知あるいは未対応の脆弱性に対応を求められても、実際には対応できるものではない。未知・未対応の脆弱性に対してできることと、その手順は以下のようなものとなる。

① セキュリティ対策すべき資産を特定する
② 特定した資産に対しての脅威を分析する。
③ 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する。
④ リスクの重要性に応じて何らかの対策を取る。

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、未知の脆弱性のリスクは予測可能性が低く、具体的な対応箇所が明確にならないことも多い。そのため普通は、影響を軽減することにとどまってしまう。さらに、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。

未知・未対応の脆弱性対策は、可能性をある程度低減する、被害範囲をある程度狭める、早期に発見できるようになるだけ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効果がある。

また、脆弱性に対する対応漏れも、未知の場合と同様に考える必要がある。漏れなく完璧に遂行することができるのが理想だが、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、次に記載する保険的対策をしておくことが必要になる。

2-4. セーフティネットとしての保険的対策

「保険的対策」とは、「攻撃による影響を軽減する対策」をいう。根本的解決とは異なり、脆弱性の原因そのものをなくすものではないが、攻撃から被害までの各フェーズにおいて、それぞれの影響を軽減することができる。

●攻撃される可能性を低減する

例）攻撃につながるヒントを与えない

●攻撃された場合に、脆弱性を突かれる可能性を低減する

例）入力から攻撃に使われるデータをサニタイズ、無効化する

●脆弱性を突かれた場合に被害範囲を最小化する

例）アクセス制御

●被害が生じた場合に早期に知る

例）事後通知

保険的対策は脆弱性の原因そのものをなくす対策ではないため、保険的対策のみに頼る設計は推奨されない。あくまでも開発の設計段階から、根本的解決の手法を採用することを優先すべきだ。

かといって、保険的対策が無用というわけではない。保険的対策は、根本的解決の実装に漏れが生じた場合の「セーフティネット」として機能する。根本的解決と保険的対策を併せて採用することが有効な場合もある。

すでに開発を終えた運用段階のシステムにおいて、後から脆弱性対策を実施する場合においても、根本的解決の手法を採用することが望ましいが、費用や時間、その他の事情により速やかな実施が難しい場合に、保険的対策は暫定対策として機能する。

一方、保険的対策は、対策の内容によっては本来の機能を制限する可能性もある。副作用の影響も考慮し、保険的対策を用いる必要がある。

3. システム防御のための基本原則

3-1. 脆弱性を作り込まないためのセキュリティ原則

ここでは、脆弱性を作り込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述べる。

個々の事象をすべて明確にして、その対策をあげることは現実的ではない。そこで役立つのが原則を設けることである。必ずしも個別に明確に決められていない場合にも、原則を定めてそれに従うことで、対策が行えるように準備することができる。それらはいずれも、本章 1-3「情報セキュリティリスク」で述べた情報セキュリティを守るための機密性・完全性・可用性の 3 要素と関連するものとなる。

原則の数はできるだけ少なく、さらにあまり頻繁に変える必要のないものであることが望ましい。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、たとえば次のものを参照してみることをすすめる。

・SANS の Principles of secure design
https://www.sans.org/reading-room/whitepapers/application/secure-design-exploitinfusion-35587

・Build Security In の Design Principles
https://buildsecurityin.us-cert.gov/articles/knowledge/principles/design-principles

・Oracle linux の Design Principles for Secure Coding
https://docs.oracle.com/cd/E37670_01/E36387/html/ol_desprinsc_sec.html

・OWASP Secure Coding Principles
https://www.owasp.org/index.php/Secure_Coding_Principles

・NIST SP800-27

http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf

・IATAC Software Security Assurance1

http://www.dtic.mil/dtic/tr/fulltext/u2/a472363.pdf

3-2. 設計時の原則に関して

設計原則については、Saltzer & Schroeder の 8 原則[4]を取り上げる。1975 年にセキュリティメカニズムの設計と実装のためにまとめられたもので、今なお多くの設計原則でも参照されている。単純さと制限のアイデアを採用しており、すべての項目が機密性、完全性、可用性に関連している。

＜Saltzer & Schroeder の 8 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。

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**

一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ほうが強固だし柔軟でもある。

[4] BASIC PRINCIPLES OF INFORMATION PROTECTION
http://web.mit.edu/Saltzer/www/publications/protection/Basic.html

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ーは、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

複数のユーザーが共有し依存する仕組みの規模を最小限に抑える。

８．Psychological acceptability：心理学的受容性

ユーザー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をする。

以上の原則の訳は「C/C++セキュアコーディング」(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）の「8.1 安全なソフトウェア開発のための原則」から引用している。

以下に各原則の概要を記述する。

●設計原則１：Economy of mechanism 効率的なメカニズム

単純明快で小さな設計を心がける。大きく複雑なものは間違いやすく、それゆえに経済的ではなくなる。設計と実装が単純で小さければ、エラーの可能性は少なくなる。安全な開発とは、攻撃対象の領域を減らすことで全体的なリスクを低減させることだ。「KISS」の原則、“Keep it short and simple”または“keep it simple, stupid”である。この原則は、セキュリティメカニズムは可能な限り単純かつ小さくするべきであるとしている。

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

誤操作・誤動作による障害が発生した場合でも安全に処理する、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。

禁止を基本とすることで、アクセス権やセキュリティ関連の属性が明示的に付与され

ていない場合は、常にそれを拒否することが可能になる。
さらに、サブジェクトがそのアクションまたはタスクを完了できない場合は、終了する前に、システムのセキュリティ状態に加えられた変更を元に戻さなければならない。こうすれば、たとえプログラムが失敗してもシステムは安全を保つことができる。

●設計原則 3 ：Complete mediation 完全な仲介

完全な仲介により、セキュリティメカニズムが漏れなく適用されるようにする。すべてのオブジェクトへのアクセスが保護スキームに準拠しているかどうかをチェックし、それらが許可されていることを確認する。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ことが必要だ。
考えられるアクセス方法はすべ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が必要となる。たとえば、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となる。そのため、サーバ側ですべ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 に、この原則のアンチパターンとなる、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。

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

設計はオープンにし、設計内容を秘密にしてはならない。セキュリティメカニズムの機密性によって安全性が向上することはほとんどないと考える必要がある。攻撃者は分解や分析などの技術的手段、あるいはソースコードリストをゴミ容器から探すなどの非技術的手段（「ダンプスターダイビング」と呼ばれる）を介して、詳細を推測することができるからだ。

また、セキュリティメカニズムは、その安全性を確保するために、攻撃者がそのメカニズムの知識を持たないという予見に頼ってはならない。防御システムそのものの内容は公開されるべきであり、メカニズムの安全性はパスワードや秘密鍵のように比較的少ない項目、そして簡単に変更可能な要素で守るようにしたほうが良い。そうすることで広範囲の第三者による検査を受けることができる。

さらに広く配布されている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。デコンパイラ（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

必要となるアクセス権を設定して、権限を分離する。さらに目的となる対象へのアクセス条件も複数に分離することが理想となる。そうすることで、もしある防御システムが破られても他の有効な条件により無制限なアクセスを許すことにならない。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。可能であれば、一つの鍵で保護する仕組みよりも二つの鍵を使って保護する方が強固でありかつ柔軟でもある。

●設計原則６：Least privilege 最小限の権限

ユーザーやプログラムには必要となる権限のみを与え、管理者以外は特権を持たせない。システムのすべてのユーザーとすべてのプログラムは、業務を遂行するために必要な最小の権限の組み合わせにより操作を行うようにする。

また、割り当てられている権限以外のアクセス権がどうしても一時的に必要となった場合、対象者の追加の権利は行為完了後直ちに破棄するようにする。ユーザーには常にプログラムの最小限の部分にだけ、必要なアクセス権を持たせるようにする。

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

複数のユーザーが共有する共通メカニズムは必要最小限にし、ユーザーとそのアクティビティを互いに分離する。

ユーザーはプロセスやスレッドを共有するべきではなく、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おり、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

セキュリティメカニズムは必要であるが、それによりリソースにアクセスしにくくするべきではなく、開発者だけでなくユーザーにも受け入れられるように設計することが重要になってくる。

セキュリティ強化だけを考えた結果、使いにくいシステムになっては使われない。使いやすいシステムはユーザーに避けられることがなく、ユーザー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必要だ。同様に、セキュリティ関連のユーザープログラムも使いやすく、わかりやすいメッセージを出力する必要がある。たとえば、パスワードが拒否された場合、パスワード変更プログラムは不可解なエラーメッセージを表示するのではなく、パスワードが拒否された理由が簡単にわかるように説明する必要がある。セキュリティの仕組みがユーザーの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。

3-3. 実装時の原則に関して

実装原則には、開発の一般原則を明確に記した SEI CERT Top 10 Secure Coding Practices（https://www.securecoding.cert.org/confluence/display/seccode/Top+10+S

ecure+Coding+Practices）をあげることにする。原文で 7 番目にある 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 Validate input.と対に意識してもらうために、ここでは 2 番目に置いた。

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）[5]］

※2［Saltzer, J.H."Protection and the Control of Information Sharing in Multics." Communications of the ACM 17, 7 （July 1974）：388-402.[6]、
SaltzerJ.H.& Schroeder, M. D. "The Protection of Information in Computer Systems." Proceedings of the IEEE63,9（September 1975）,1278-1308.[7]］

[5] http://resources.sei.cmu.edu/asset_files/BookChapter/2005_009_001_52692.pdf

[6] CSE 221：Graduate Operating Systems
https://cseweb.ucsd.edu/classes/wi08/cse221/papers/saltzer74.pdf

[7] I. BASIC PRINCIPLES OF INFORMATION PROTECTION
http://web.mit.edu/Saltzer/www/publications/protection/Basic.html

●実装原則 1 ：Validate input.

信頼できないデータソースからの入力をすべて検証する。適切な入力検証により、ソフトウェアの脆弱性の大部分を排除することが可能になる。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、ユーザー管理のファイルなど、外部データソースはそのほとんどを疑う必要がある。

DBMS から取得したものについても同様だ。とくに Web は、ユーザー用フォームからの入力に加え、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必ず精査を行う。未チェックのデータフローは脆弱性の最も一般的な原因の一つとなる。

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

コマンドシェル、リレーショナルデータベース、および市販の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-C：

JPCERT コーディネーションセンター「STR02-C. 複雑なサブシステムに渡すデータは無害化する」（https://www.jpcert.or.jp/sc-rules/c-str02-c.html）

攻撃者は SQL、コマンドなど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可能性がある。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ため、この問題は必ずしも入力検証の問題ではなく、出力側の問題となってくる。なぜなら通常使用される機能は、呼び出しが行われる際のプロセスとなる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るからだ。

外部へのデータは渡した先で問題を起こさないように加工する必要がある。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行う。たとえばデータを渡す先が Web ページへの出力部分ならば XSS 対策をする必要があるし、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。
データを渡した先で問題を起こさないようにする対策は、不正文字の無害化以外にもある。データ型に応じたデータの取り扱い、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したい。

●実装原則 3 ：Heed compiler warnings.

プログラムコードに対して行われる最初の精査行為であるコンパイラの警告に注意することが必要だ。コンパイラで利用しうる最高の警告レベルを使用してコードをコンパイルし、コードを変更することによって警告を排除する。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可能になる。

・C MSC00-A／C++ MSC00-A：
Carnegie Mellon University
「MSC00-C. Compile cleanly at high warning levels」
https://www.securecoding.cert.org/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels
JPCERT コーディネーションセンター
「MSC00-C. 高い警告レベルでのコンパイルで警告が出ないようにする」
https://www.jpcert.or.jp/sc-rules/c-msc00-c.html

コンパイラからの警告は有効な情報だ。しかしながら、警告レベルによってはそのままでも実行可能だったり、非常に多くの似たような情報が出力されたりするので開発者は無視してしまう場合がある。できる限りコンパイラだけでなく、静的および動的解析ツールを併せて使用したい。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可能になる。

●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。セキュリティは守るべきもの

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ものではない。システムにおいては、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義になる。

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従って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。

●実装原則 5 ：Keep it simple.

設計原則 1 の「Economy of mechanism：効率的なメカニズム」と同意で、デザインはできるだけシンプルに小さくする必要がある。複雑な設計では、実装や使用方法に誤りが生じる可能性が高くなる。さらに、セキュリティメカニズムがより複雑になるにつれて、適切なレベルの保証を達成するために必要な労力が劇的に増加する。

同じ結果を得られる実装は一つとは限らない。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だ。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作業全般で、ミスを犯す可能性を低くすることができる。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る。

●実装原則 6 ：Default deny

設計原則 2 の「Fail-safe defaults：フェイルセーフなデフォルト」と同意で、アクセス決定は許可ベースではなく拒否をデフォルトにする。

ここでの原則は、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している。

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

どのようなプロセスも実行するためには、必要最低限の特権セットで実行すべきであり、最小権限の原則に従う。また、設計原則 6 の「Least privilege：最小限の権限」と

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

たとえば、システム設定を行うためにアプリケーションが管理者権限で実行され、これらの権利がすぐに放棄されず、攻撃者がデータを転送する隙をついた場合、攻撃者は昇格した権限により任意のコードを実行する可能性が生まれてしまう。

●実装原則 8 ：Practice defense in depth.

設計原則 5 の「Separation of privilege：権限の分離」と同意で、対象へのアクセスに対して複数の条件を設定した多層防衛を行う。

これは防衛の一つの層が不十分であっても、別の防衛層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、脆弱性が悪用された場合でも影響を制限したりすることが目的だ。たとえば、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脆弱性が運用環境で悪用される可能性を減らせる可能性がある。

また防衛策も、根本的対策だけでなく、異なるタイプとなる保険的対策も含めたものを行うようにしたい。これにより、一つの対策が不完全であったり、攻撃者に破られたりしてもすべてを失うのではなく、別の対策により被害をある程度限定できるようにすることが可能になる。つまり、攻撃者が一つのハードルをクリアしたとしてもアクティブな状態にはなっておらず、完全にアクティブになるためには複数のハードルを克服しなければならないことを意味する。

●実装原則 9 ：Use effective quality assurance techniques.

優れた品質保証技術は脆弱性の特定と排除にも有効であるため、効果的な品質保証テクニックを利用することが重要だ。外部のレビュアーなど独立したセキュリティレビューは、より安全なシステムにつながる。

Fuzz テスト、侵入テスト、ソースコードの監査はすべて、効果的な品質保証のプログラムの一部として、実装前もしくは実装段階で組み込まれる必要がある。

●実装原則 10：Adopt a secure coding standard.

セキュアコーディングのため、ターゲット開発言語やプラットフォームは標準を適用し、開発を行っていく。セキュリティ対策の実施は、複数の箇所で同じような作業を行うことになる。そのため、標準を採用することで共通した作業が行えるようになり、効率的な対応が可能になる。

実施した対策では不十分な部分があったことが見つかったとしても、標準的な共通作業をベースにした対応となるので、比較的簡単に修正することができる。

第 2 章

効果的なセキュリティ戦略

第 2 章　効果的なセキュリティ戦略

1. セキュリティリスクを識別する脅威モデリング

1-1. セキュリティ対策の優先順位付け

本章では第 1 章でも取り上げたリスク＝【情報資産】×【脅威】×【脆弱性】の考え方に従って、ソフトウェアが受けやすい、より高いリスクを識別し、セキュリティへの取り組みに対する適切な優先順位付けを行う脅威モデリングの手順について述べる。

脆弱性とは、悪用を可能にするシステム内の弱点である。これに対し脅威とは、情報資産に対して害をもたらす可能性のある事象を意味する。たとえば、ソフトウェアの不具合などのシステムリスク、機器の故障や障害、内外からの攻撃といった運用リスク、さらに地震や災害などの環境リスクも含まれる。

今日、ソフトウェアシステムはさまざまな脅威にさらされており、脅威の数も増大している。しかしながら、開発対象のネットワークシステムが存在するすべての脅威に遭遇するわけではなく、また、すべての脅威に対して高いレベルの対策をとることは現実的に不可能である。脅威モデリングは、どこに集中して防御対策すべきかを判断するうえでも非常に有効である。

1-2. 脅威モデリングをいつ行うか

脅威モデリングとは、情報資産を意図的にリスクにさらすことで、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っているかを洗い出す行動をいう。

脅威モデリングのメリットの一つは、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ことだ。そのため、潜在するセキュリティリスクをソフトウェア設計段階の比較的初期に見つけ出すことが可能となり、効果的にリスクを制御することができる。

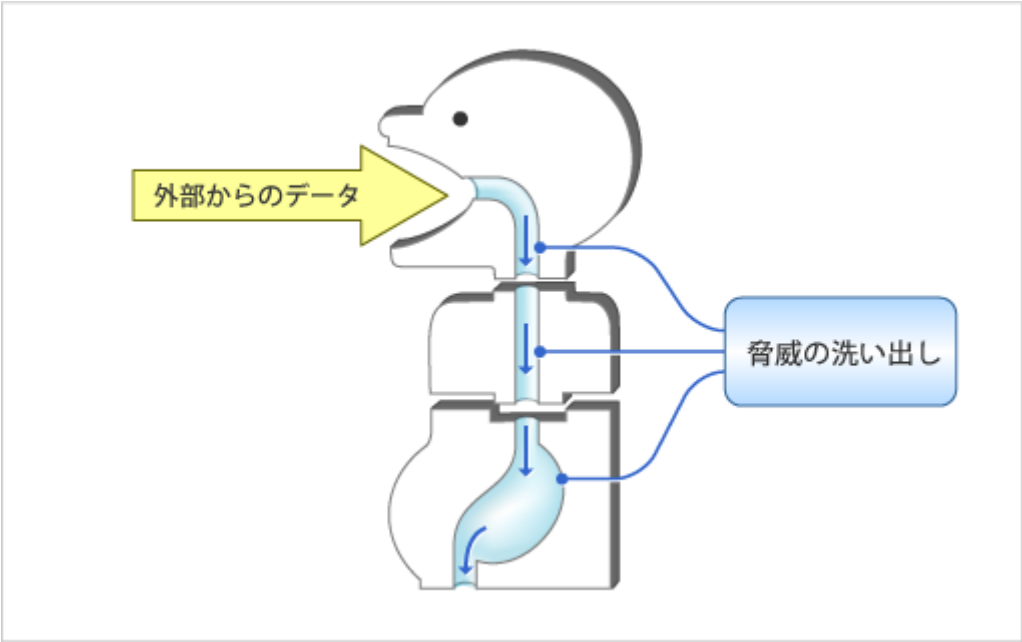
具体的には、設計の概要がある程度落ち着いて、次にあげる事項の見通しがはっきり

してきた時点から脅威モデリングは行うことができる。

●開発対象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
●データはソフトウェアシステム内部をおおむねどのような形で流れるか

以降も、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。

図解 2-1　脅威モデリング



出所：独立行政法人情報処理推進機構（IPA）
ISEC セキュア・プログラミング講座

1-3. STRIDE 手法による脅威の分類

脅威モデリングにはさまざまな手法がある。その一つであるマイクロソフト社の Security Engineering and Communications グループが策定した STRIDE 手法は、次の

6 つの事象が起きた場合を具体的に想定することで、実際の脅威を識別しやすくしているのが特徴である。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（http://www.jnsa.org/active/houkoku/web_system.pdf）においても、3 案の一つとしてこの STRIDE を用いている。

＜STRIDE 手法＞

S：なりすまし（Spoofing Identity）

悪意のある第三者が正規のユーザーになりすましてシステムにログインし、権利を持たない者による保護すべき資産の改変、また攻撃者が他のユーザーを装ったり、悪意のあるサーバが有効なサーバであるかのように装ったりすること。

T：改ざん（Tampering with data）

権利を持たない者による、保護すべき資産の改変。悪意をもってデータベースに記録されている情報を書き換えること。

R：否認（Repudiation）

サーバへのアクセスログを消去して侵入の痕跡を消し、行った操作の事実を否定すること。利用者による組み込みシステムや組み込みシステムに提供されるサービスの利用とその否定も含まれる。否認不能性の実現はもともとそれなりに手間であり、ユーザー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ようにすることが前提になる。

I：情報の漏洩（Information Disclosure）

データベースから顧客情報が流出するなど、不本意な相手に情報が開示されてしまうこと。権利を持たないものによる、秘匿された保護すべき資産の入手。

D：サービス妨害（Denial of Service）

サーバにアクセスを集中させ、正当なユーザーの要求の処理ができなくなるなど、

サービスの提供を行えなくすること。権利を持たない者による、保護すべき資産の利用の妨害。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ーが、高い権限を取得すること。これにより、サーバ内で不正なプログラムの実行等が可能になる。

1-4. 脅威モデリングのプロセス

脅威モデリングは、製品、アプリケーション、ネットワーク、環境にもたらされるセキュリティのリスクを判断し、どのように攻撃が発生するかを認識するのに役立つ。このモデリングにより、緩和が必要な脅威とその緩和方法を決定することを目標とする。

＜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。守るべき資産にはたとえば、クレジットカード番号などのデータ資産、知的所有権やデジタル著作権管理に基づくコンテンツなどの技術資産などがある。

②全体像の把握

脅威モデルの範囲を定義する。そのためには、開発するアプリケーションの主要な機能、特徴などを理解し、全体像を把握しなければならない。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必要がある。

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、

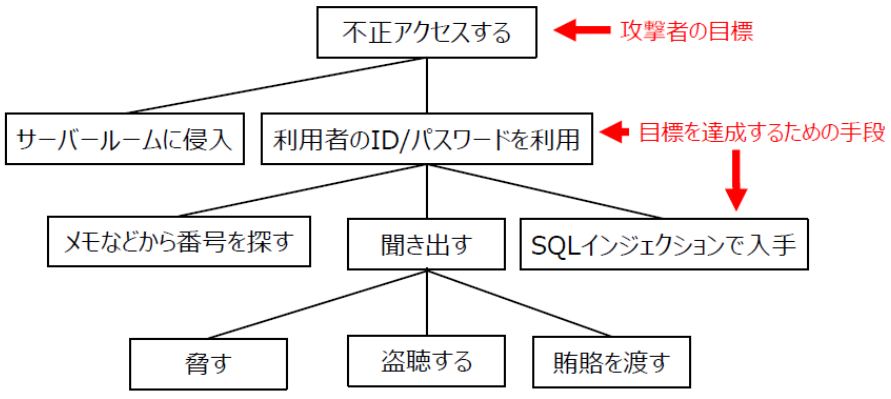
データフロー、入口、および出口を特定する。アプリケーションのメカニズムをよく理解することで、脅威を明らかにし、脆弱性を検出することが容易になる。

④脅威の特定

対策する必要があるセキュリティリスクを特定する。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。

最初に、MITRE（https://capec.mitre.org/）が公開している CAPEC（Common Attack Pattern Enumeration and Classification）などを参考に脅威についての傾向を把握し、システムの脅威を列挙する。続いて、脅威情報を分類した Attack Library を作成する。これにより、脅威に対処する作業の効率化が期待できる。
この Attack Library をもとに、攻撃者の目標と攻撃手段を洗い出した Attack Tree を作成する。Attack Tree の作成には時間がかかるが、脅威についての攻撃手段を可視化することが可能になり、攻撃されやすい箇所の特定が容易になるというメリットがある。

図解 2-2　Attack Tree の例



出所：FFRI.Inc「脅威分析の役割と手法の紹介」

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

いのであれば、はじめから脅威として取り扱う必要性がなくなる。たとえば、誰にでも公開している情報への参照で、参照したかどうかを追跡する必要がなければ、仮になりすましをされても特に問題にはならない。 そのため、本当に困るものを開発対象のソフトウェアシステムに対する脅威として取り上げ、Attack Tree を作成していくことが効率的だ。

＜脅威の例＞

●攻撃者がこの項目（ユーザーやサーバ）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

⑤ リスクの格付け

収集した情報をもとに、リスクの評価を行う。

原則として、最も重大なリスクを最初に解決する必要がある。たとえ簡単もしくは安価に解決することができるとしても、重要性の低いリスクから解決しても総合的なリスク管理の役に立たない。そのため、脅威の実現可能性や保護資産にもたらされる被害規模を検討し、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ようにする。

また、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る。ランク付けをするには、上記で洗い出した脅威の一つ一つに関して、それぞれに大・中・小のおおまかな評価を下し、総合評価を脅威のランクとして割り当てる方法がある。

次の項目が大・中・小の尺度となる。

●被害程度・・・侵害により生じる被害の程度

●再現性・・・・脆弱性が利用できる頻度

●容易さ・・・・この侵害を実行する難易度

●被害者数・・・被害を受ける人の数の規模

ただし、攻撃手順がわからなければ「再現性」と「容易さ」を判断することは難しい。可能性を考えるという意味では、次の問いを考えてみる方法もある。

●攻撃者はこの項目（ユーザーやサーバ）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大まかに評価した大・中・小から、事前に方針として「中」または「大」を選択すると決めておくといいだろう。以前は、常に「大」を選択することが推奨されていたが、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。

2. セキュリティ品質を重視した開発プロセス

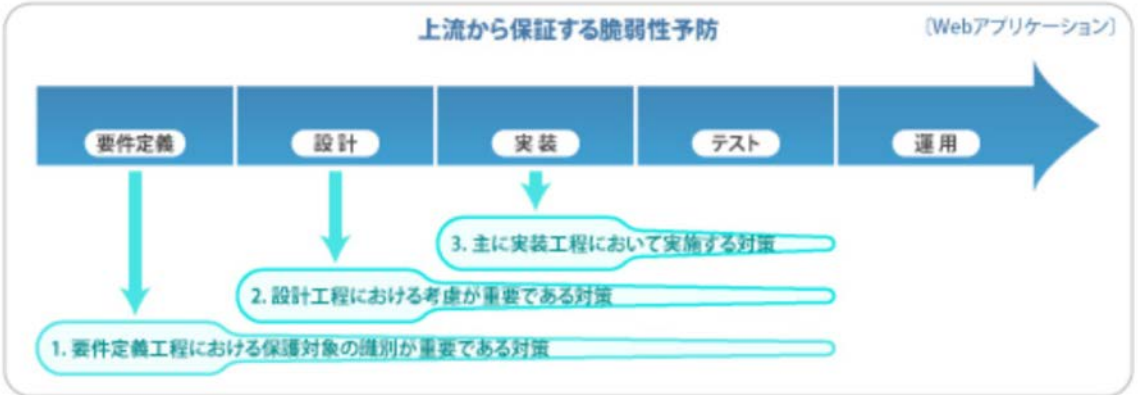
2-1. どの工程からセキュリティ対策に取り組むか

セキュリティ品質を重視すると、脆弱性等への対応は開発プロセスの上流工程から実施されるのが望ましい。

セキュリティはこれまで、システムテストの一環として確認するケースが多く、後からの対策として行われがちであった。しかしながら、情報漏洩やデータの改ざん、システムの停止など、実際に発生するインシデントを分析すると、要件定義や設計といった上流工程に問題があることが多い。また、テスト工程に至ってから脆弱性対策を行うことは、いったん作ったソフトウェアを後から修正することになるため手戻りが多く、品質、コスト、納期の点においても大きな不利が生じる。

上流工程にこそ、セキュリティを実現する重要な要素がある。「早めの 3 工程によるセキュリティ品質確保」（独立行政法人 情報処理推進機構セキュリティセンター2010 年 8 月公開）においても伝えているように、セキュリティは品質と同様に開発の始めから作り込むものと意識変革する必要がある。最近では、内閣サイバーセキュリティセンター（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。

図解 2-3　上流から保証する脆弱性予防

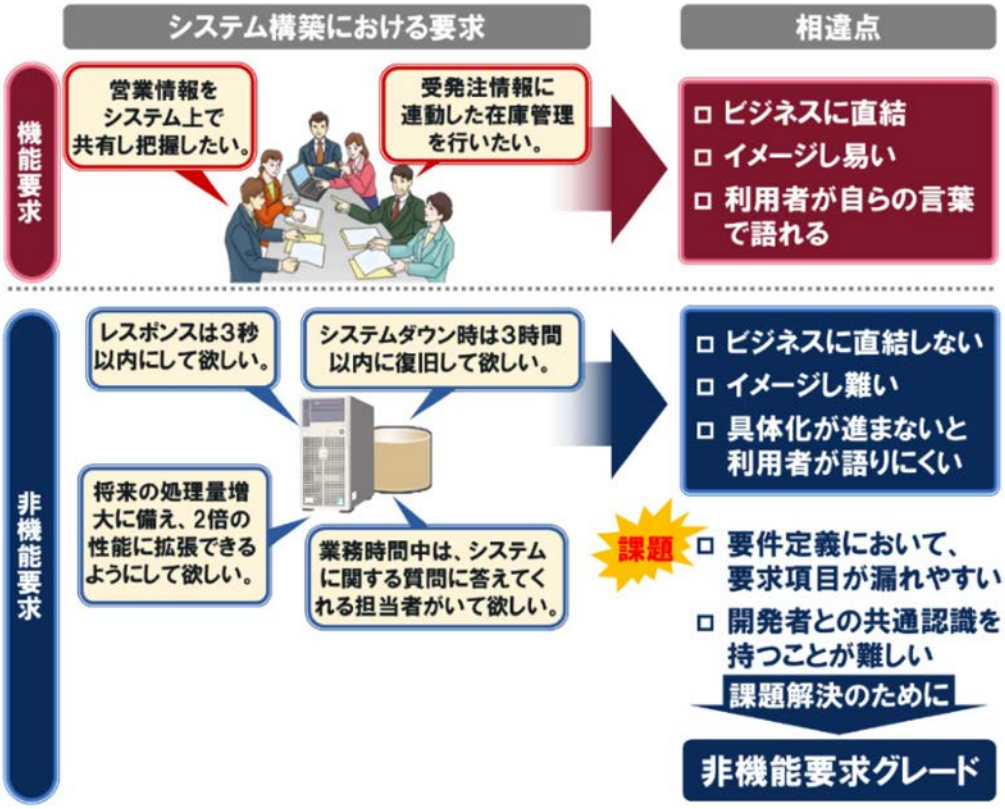


出所：独立行政法人情報処理推進機構（IPA）
「早めのチェック　3 工程によるセキュリティ品質チェック」

また、情報処理推進機構（IPA）が公開する「非機能要求グレード 2018」（https://www.ipa.go.jp/sec/softwareengineering/std/ent03-b.html）においてセキュリティは、非機能要件として非機能要求グレードの 6 大項目の一つとして取り扱われている。

非機能要件とは、ハードウェア、OS、ミドルウェア、データベースといった「システム基盤」の強度や品質に関する要求、ユーザインタフェースの操作性やソフトウェアの品質など「アプリケーション」に対する要求などをいい、Web アプリケーションの機能に対する要求である機能要件と並んで必要不可欠な要素である。しかしながら、業務と直結する機能要件と比較して、非機能要件は明確な要件が出にくい傾向がある。基準がないため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といったことが起きている。

図解 2-4　機能・非機能要求の相違点と課題



出所：独立行政法人情報処理推進機構（IPA）
「システム構築上流工程の強化（非機能要求グレード）」

セキュリティをまったく行わないのは論外にしても、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を無効化しなければならなくなることにもなりかねない。セキュリティも他の要件と同様、最高レベルではなく、適切なレベルで行うことを意識する必要がある。

適切に行うためには、何を実施するかを明確にすることが重要になってくる。逆に、実施しないことに対しては、事前に明確な合意が必要になる。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となり、この既知の脆弱性を含めて、発注者側と共通の認識を持っておくことが必要となる。

ソフトウェアのセキュリティ品質を確保するためには、次の 3 つのアクティビティが重要となってくる。この 3 つについてそれぞれ、次の項から説明していく。

①セキュリティレビュー

設計工程において、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

②ソースコードレビュー

実装工程において、記述したソースコードを直接検証する

③セキュリティテスト

テスト工程において、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

2-2. 設計工程におけるセキュリティレビュー

セキュリティレビューを行う目的は、セキュリティ対策に漏れがないかを早期に見つけ出し、設計者へフィードバックすることにある。

(1) 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

次のようなものである。

・セキュリティ要件定義書

・ソフトウェア構造設計書

・業務仕様設計書

・モジュール分割設計書

・テスト計画書

(2) 要領・観点

必要とされるセキュリティ要件により異なるが、セキュリティポリシーを満たし、セキュリティ要件に対する対策と既知の脆弱性対策が行えているかどうかが重要になる。セキュリティレビューで考慮する項目は、たとえば、以下のような項目が考えられる。

●上流設計

・本人認証のロジックは堅固か

・エラーメッセージは関係者以外にわかりづらいようにしているか

・アクセス承認のロジックは迂回しにくいものになっているか

・情報出力箇所で不用意な重要情報の出力は避けているか

・ログ記録による証跡確保を実施しているか

・ログそのものの漏えい対策は十分か

●ファイルおよびプログラムモジュール

・侵害されると重大な結果を招くプログラムを chroot、jail 等の環境で稼働させるようにしているか

chroot：Unix システムコールの一つで、ファイルシステムのルートディレクトリの位置を変更するコマンド。コマンドを任意のディレクトリの下に閉じ込めてファイルシステムのアクセスに制約をかけることができる。chroot から実行されたコマンドは、指定されたルートディレ

クトリよりも上の階層へアクセスすることができない

jail：chrootの機能を拡張して、ネットワーク・プロセス等も隔離できるようにしたFreeBSD特有の仕組み。ファイルシステム空間、プロセス空間およびネットワークを隔離する技術である

・起動された子プログラムが親プログラムを侵害することを想定し、対策を講じているか

・秘密保持が必要なパラメータをコマンドライン引数で受け渡さないようにしているか

・秘密保持が必要な情報を、ソフトウェアを構成するプログラムファイルや設定ファイルに簡単に読み出せる形で埋め込まないようにしているか

・ファイルへのアクセス権の付与を最小限にしているか

・テンポラリファイルへの他者からの干渉を避ける対策を講じているか

●連続稼働と動作タイミング

・メモリリークが想定され対策されているか

・サービス不能状態が想定され対策されているか

・プログラムの稼働途中に短いタイミングであっても侵害に無防備な状況が生じる問題が想定され、対策が講じられているか

2-3. 実装工程におけるソースコードレビュー

ソースコードレビューはソフトウェアレビューの一種で、開発担当者が書いたソースコードが読みやすくわかりやすく書かれているかを確認し、さらにはプログラム中のセキュリティ脆弱性あるいはその兆しを検出することが目的となる。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、速やかにソースコードの修正を行っていく。

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プロジェクト

の場合はとくに、後々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従って一定の品質であることが非常に重要な要素の一つとなってくる。それがソースコードレビューを行う理由でもある。

以前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が普通に先行していたが、現状のように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においてセキュリティが取り扱われたことはほとんどなかったと考えられる。

(1) ソースコードレビューを行う人物

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ー

開発者本人がソースコードを見直すだけでなく、複数の異なる立場や視点からのチェックの双方が行われるのが理想となる。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が身につけられること、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などがあげられる。

ペアプログラミングという方法を用いることで、常に複数人の目を通して改善を進めることもできる。ただし、この方法は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。

ソースコードレビューを行うタイミングは、ソースコードが新たに書き起こされるか、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が望ましい。

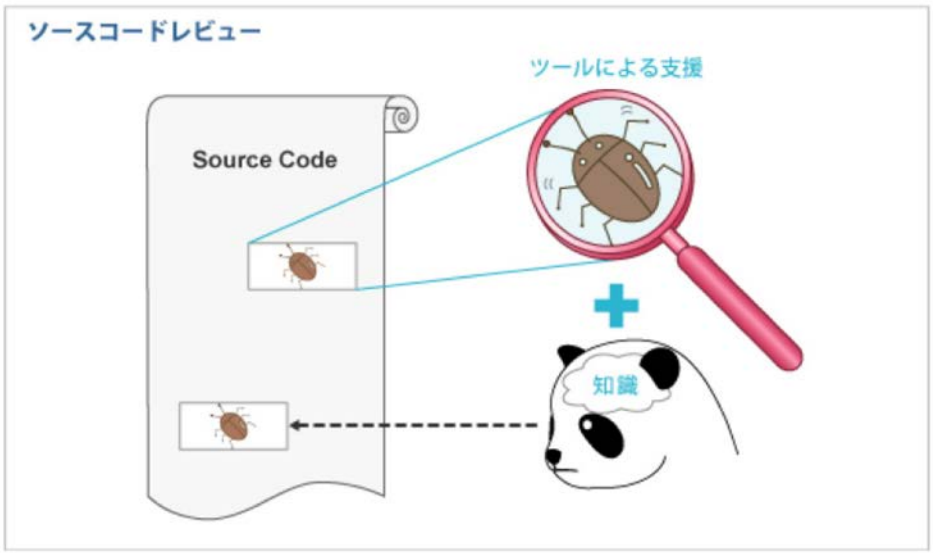
(2) 要領・観点

レビューでは、脆弱性検出のために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従っているか？
・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ところはないか？
・設計では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？
・情報漏洩やアクセス制御の迂回が起こらないか？
・領域からデータがあふれる可能性があるところはないか？

レビューを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つけるという方法がある。さらに 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。

図解 2-5　ソースコードレビュー



出所：独立行政法人情報処理推進機構（IPA）
「セキュア・プログラミング講座」（2016 年版）

ソースコードレビューは、予算や開発期間の制約等により不十分になることがある。その場合、ブラックボックステストでは見つけられない脆弱性を残してしまう可能性が高くなる。最低でも重大なリスクが想定されるコードについては重点レビューを行い、コード検査を自動化するなどの対処を行う必要がある。

2-4. テスト工程におけるセキュリティテスト

セキュリティテストは、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が目的となる。悪意のある攻撃や通常のソフトウェア障害に直面しても、システムを保証することができることが望ましい。

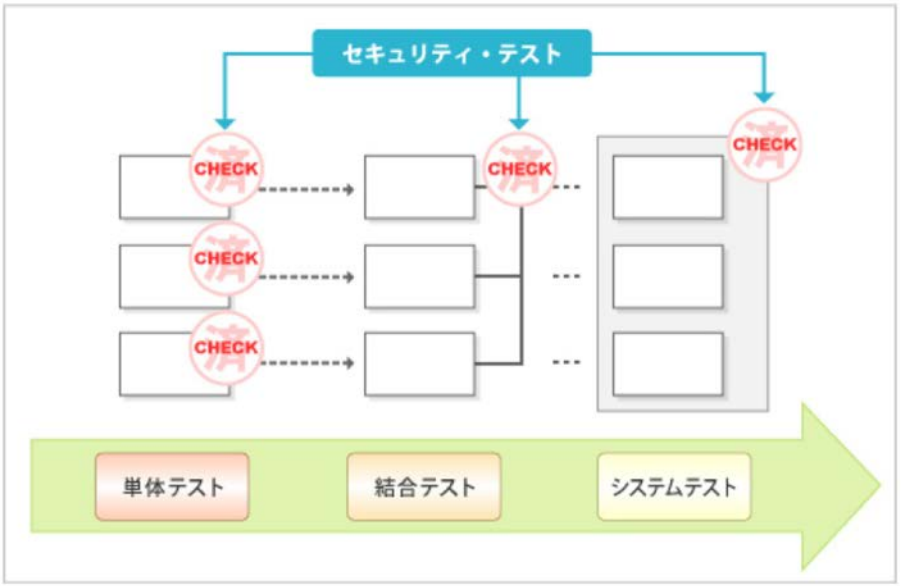
（1）要領・観点

・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

図解 2-6　セキュリティテスト



出所：独立行政法人情報処理推進機構（IPA）「セキュア・プログラミング講座」（2016 年版）

(2) 方法

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、実装漏れがないか、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。

セキュリティテストは、通常のテストと同様に次の 2 種類の方法がある。

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、プログラムの内部構造とは関係なく、仕様を満たしているかどうかのみを検証する技法となる。そのため、外部からシステムの機能をテストする。

セキュリティテストで行うブラックボックステストも同じで、外部からの入力によるプログラムの動作を確認する方法で行う。

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こともできる。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。

＜ツールの例＞

●領域あふれを検出するツール

・madflap（-fmadflap -lmudflap、GCC のオプション）

・Valgrind の memcheck ツール（Linux で稼働するヒープデバッガ）

・/RTCs （Visual C++のコンパイルオプション）

●32/64 ビット符号付き整数の演算時のオーバーフローの検出

・-ftrapv （GCC のオプション）

●ビット幅が小さい整数変数への代入において上位ビットが失われることの検出

・/RTCc （Visual C++のコンパイルオプション）

2）ホワイトボックステスト

システムの内部構造を考慮しないブラックボックステストに対し、システムの内部構造をテストするのがホワイトボックステストとなる。ホワイトボックステストとは一般的に、システム内部の構造を理解したうえで、ロジックや制御の流れが正しいかどうかを検証する。

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。データフロー、制御フロー、情報フロー、コーディング慣行、およびシステム内の例外およびエラー処理の分析が含まれる。

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、導入が難しい場合がある。

2-5. セキュリティテストの注意事項

セキュリティテストを行うにあたり、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。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すべて漏れなくテストしようとすると膨大な量になってしまう。そのため、要点を絞って行うことが重要だ。また、同様の項目があることを理由に、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。

なお、セキュリティテストにおける確認作業は、その確認すべきそもそもの実作業が確実に実施されていることを前提にしている。実作業が要件を満たした成果物となっており、そのうえでセキュリティを担保していることが大切だ。

第 3 章

ネットワークシステム構築における
セキュリティ要件の概観

第 3 章　ネットワークシステム構築におけるセキュリティ要件

1. システム開発で計画すべきセキュリティ機能

1-1. セキュリティ機能の基本的な考え方

必要なセキュリティ機能はアプリケーション毎に異なるが、中には共通して必要とされる機能もある。共通して必要とされる機能については複数のアプリケーションで共有・共用するほうが良い場合もあり、必ずしもアプリケーション毎に開発されるべきものではない。

本章では、セキュアなネットワークシステムを構築する際に一般的に必要となるセキュリティ機能を取り扱う。これは、第 1 章「3-2. 設計時の原則に関して」で概説した設計原則における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機能となる。また、第 1 章「1. セキュアなネットワークシステム構築の基本的な考え方」で取りあげたリスク＝【情報資産】×【脅威】×【脆弱性】の脅威部分への対策として作成される。

ここでは、次の 7 つの機能を取り上げる。なお、マイクロソフトの「Web アプリケーションの脅威モデル」におけるセキュリティフレームでは、以下 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 つも含まれる。

①認証
②認可
③暗号化
④入力の検証
⑤出力のエスケープ
⑥例外処理
⑦ロギング

これらの機能は一般的に、セキュアなネットワークシステムを構築するうえで欠かすことができない。したがって、システムのアーキテクチャを検討する段階で取り入れら

れる必要がある。

次の節から取り上げる各機能についての記述は主に、JPCERT コーディネーションセンター（JPCRET/CC）による日本語版もある「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）」(https://www.jpcert.or.jp/securecoding/OWASP_ASVS_20160623.pdf)を参考にした。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。ただし、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。

1-2. 各セキュリティ機能の役割

1-2-1. 認証

認証は、アクセスしてきている対象が本物・あるいは本人であることを確認する手続きであり、ログインページなどで一般的に用いられている。

認証には複数の方式があり、いずれの方式も ID など「対象を一意に識別する識別情報」とパスワードなど「対象しか提示できない秘密情報」の二つを対象から受け取って判定を行っている。その際、ユーザーしか知らないパスワードは入力フィールドにそのまま表示されないようにするなどの機能も求められている。

認証とは、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。確認のための秘密情報は安全に転送されなければならない。また、秘密情報は一つとは限らず、たとえば本人しか知らない情報、本人の持ち物、本人の属性など、種類の異なる複数要素を組み合わせて確認する多要素認証が採用されるケースも増えている。さらに、アプリケーション個別に認証を行うのではなく、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）

1-2-2. 認可

認可とは、認証の後にリソースへのアクセスを許可されている人だけにアクセスを許

可することをいう。ユーザーに対してログイン（認証）後、アクセスできるページやサービスなどを限定する際に用いられる。

認可とはつまり、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となる。

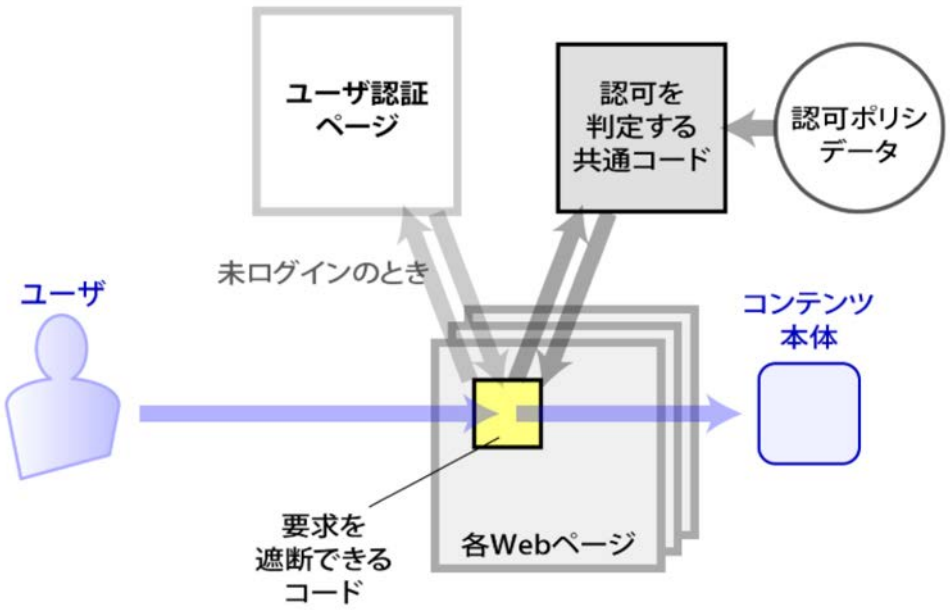
対象のアプリケーションが満たすべき要件は次の二つとなる。

・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

（参考：OWASP ASVS の V4：アクセス制御に関する検査要件）

図解 3-1　認可

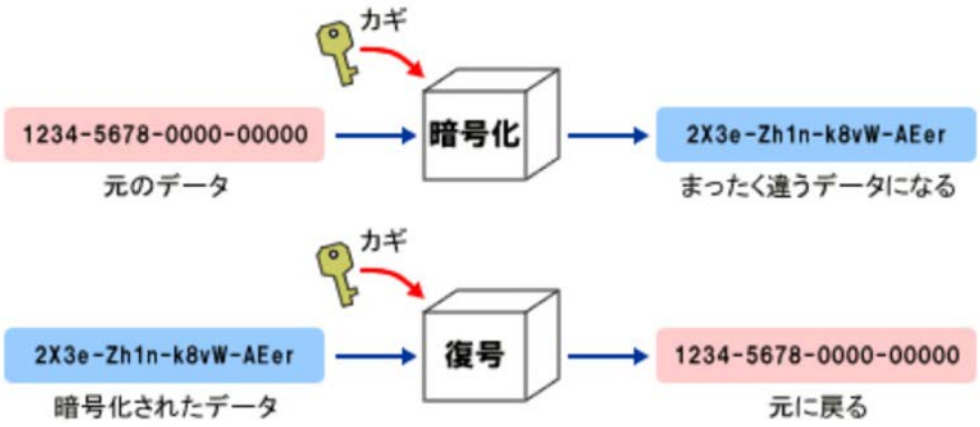


出所：独立行政法人情報処理推進機構（IPA）
ISEC セキュア・プログラミング講座

1-2-3. 暗号化

暗号化とは、情報やデータそのものを解読できないように一定の規則に従って変換（暗号化）し、暗号を解除（復号）することで情報にアクセスできるようにする方法をいう。ファイル内のデータや、ネットワーク上を流れるデータについて、その秘匿性（confidentiality）やインテグリティ（integrity）を確保するためのセキュリティ機能として用いられる。

図解 3-2　暗号化の仕組み



出所：総務省 HP「暗号化の仕組み」

暗号技術には、共通鍵暗号技術の他、公開鍵暗号技術があるが、昨今の暗号技術は両者を組み合わせて統合的に利用されることが一般化している。加えて、ハッシュ関数や擬似乱数生成技術も組み合わされて利用されることがある。たとえば、セキュア・プロトコルは鍵交換機能に擬似乱数生成技術と組み合わされて公開鍵暗号技術が使われており、暗号化機能には共通鍵暗号技術が利用されている。

暗号化機能は、理論的に安全性が確認されているものを用いることが必須となってくる。かつては開発者などが独自に作り込んだ暗号化機能が使われることもあったが、それらは必ずしも安全性が確認されたものではなかった。

また、鍵の更新は必要に応じて実施できるように、運用面についても合わせて考慮しておくことが大切となる。さらに、長期的に運用されるシステムでは、暗号方式が変更される可能性も考慮しておく必要がある。

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しなければならない。たとえば、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に注意する必要がある。

一方、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、さら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満たすべき要件は次の 3 つとなる。

・すべ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）

1-2-4. 入力の検証

入力の検証（バリデーション）は、誤った形式のデータがシステムに入力されるのを最小限に抑えることを目的に行い、開発者が想定したものだけを認めるようにすることをいう。ただし、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ことを認識しておく必要がある。

検証する際には、事前に標準化や正規化も行う。アプリケーションが受け取る入力が有効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も注意する必要がある。

Web アプリケーションセキュリティで最も一般的な脆弱性は、クライアントまたは

環境が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つながる。

対象のアプリケーションが満たすべき要件は次の二つとなる。

・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こと

（参考／OWASP ASVS の V5：悪性入力の処理に関する検査要件 ）

1-2-5. 出力のエスケープ（無害化）

エスケープはインジェクション攻撃を止めるための防御手法であり、エスケープ処理は出力先によって異なる。どこに出力しているかを明確に意識し、そこでは何がエスケープされていなければいけないかを理解しておくことが必要になる。

たとえば次のような点を意識する必要がある。

・SQL を扱う DBMS への出力であれば、SQL インジェクション対策が行われていること
・Web ページへの出力部分であれば、XSS 対策が行われていること
・HTTP レスポンスヘッダ部分への出力であれば、HTTP ヘッダ・インジェクション対策として改行コードの入力を排除していること

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。

（参考／OWASP ASVS の V6：出力のエンコード/エスケープ）

1-2-6. 例外処理

例外処理とは、業務プロセス上のある処理を実行し、処理を継続できないようなエラーが発生した際に実施する業務上の別処理のことで、設計段階で基本としたルールに従っての処理が行えない場合などへの対処となる。つまり、エラー処理の主な目標は、ユーザー、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につながる。

例外処理が占める割合は、正常系の何倍にもなることがある。さらに、例外処理は誤りや見落としも多く、不適切に処理すると、さまざまな脆弱性につながる可能性がある。最も多い一般的な問題は、エラーコードなど詳細な内部メッセージがユーザー側にも表示されてしまうことなどがあげられる。これらのメッセージは実装の詳細を明らかにすることにもつながり、攻撃者には魅力的な情報ではあるが、一般のユーザーには不要な（かつ不審な）情報となる。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、そのほとんどが一般ユーザー側には表示されるべきではない情報になる。

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 ユーザー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。

（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）

1-2-7. ロギング

ログは、ユーザー、管理者、開発者、およびインシデント対応チーム等にとって有益な形で保存されていなければならない。単に大量のログを生成することが目的なのでは

なく、有用なデータが漏れなく確実に正確に保存されており、書き換えられないようにしなければならいない。

ログは、監査や否認防止にも使用される。そのため、ログ自体が削除されたり改ざんされたりしないように不正なアクセスから守り、オリジナルのデータであることを証明できるようにシステムを保つことも必要になってくる。ログ記録では、誰が・いつ・何を行ったかを把握できるよう、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。また、ログ自体の改ざんや喪失を防ぐために別のサーバ上への保存や、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を構築することが必要な場合もある。

一方で、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い、情報漏洩リスクが高まる可能性がある。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。ログには多くの場合、センシティブな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または条例に従って保護できるように、ログとして何をどのような形で残すのかは設計段階から決めてあることが望ましい。

ログ取得で注意すべき事項には以下のようなものがある。

・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。
・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、露呈される可能性がある
・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）

2. Web アプリケーションと C/C++言語に関して

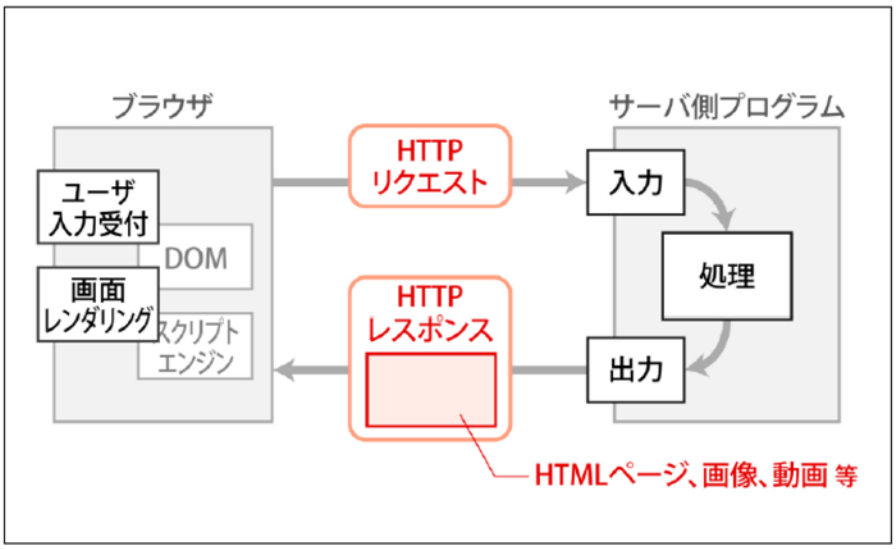
最後に、Web アプリケーションと C/C++言語の構造をみながら、その特性から生まれやすい脆弱性について記述しておく。

2-1. Web アプリケーションの特性と注意すべき脆弱性

Web アプリケーションはプロトコルとして HTTP を使用するアプリケーションであり、Web ブラウザ（クライアントサイド）と Web サーバ（サーバサイド）が緩く結合されたクライアント・サーバ構成となる。そのため、通常は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されることになる。

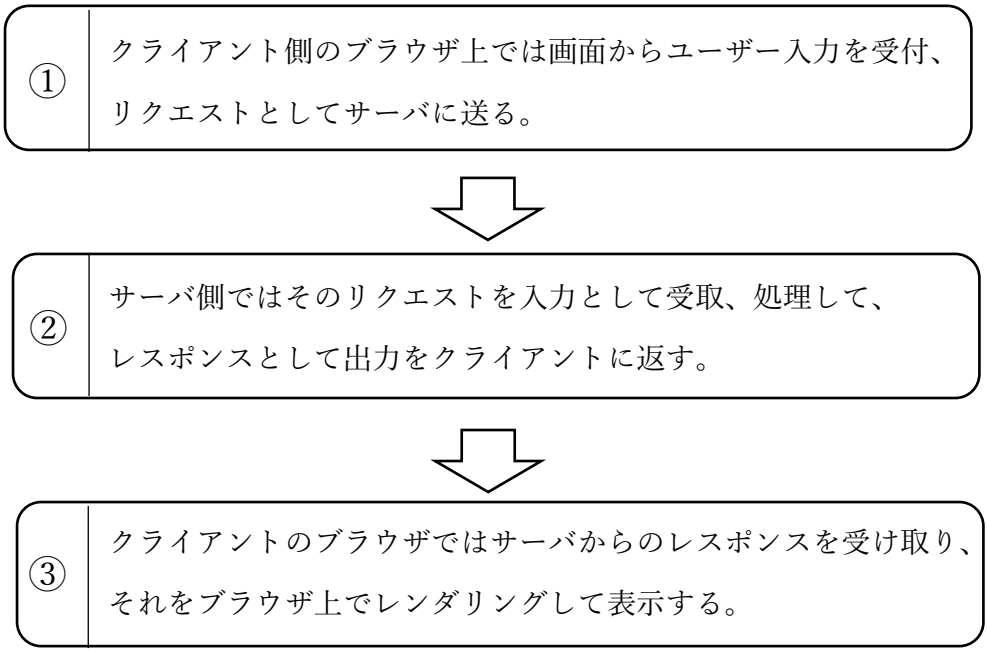
ユーザインタフェースは、サーバから供給される HTML あるいは XML の記述にもとづいて Web ブラウザで動作する。Web ブラウザの多くはネットワークアクセスを含む高機能のプログラムを解釈実行できるスクリプトエンジンを備えている。Web ブラウザと Web サーバ間の通信はプロキシサーバ等、キャッシュ機能をもつ設備によって継続される。

図解 3-3　Web アプリケーションの概略構造



出所：独立行政法人情報処理推進機構（IPA）「セキュア・プログラミング講座」（2016 年版）

Web アプリケーションは、もともとはサーバ上に格納されているファイル（情報コンテンツ）をクライアントが表示するだけのもので、以前は次に示すような非常に単純な構成だった。

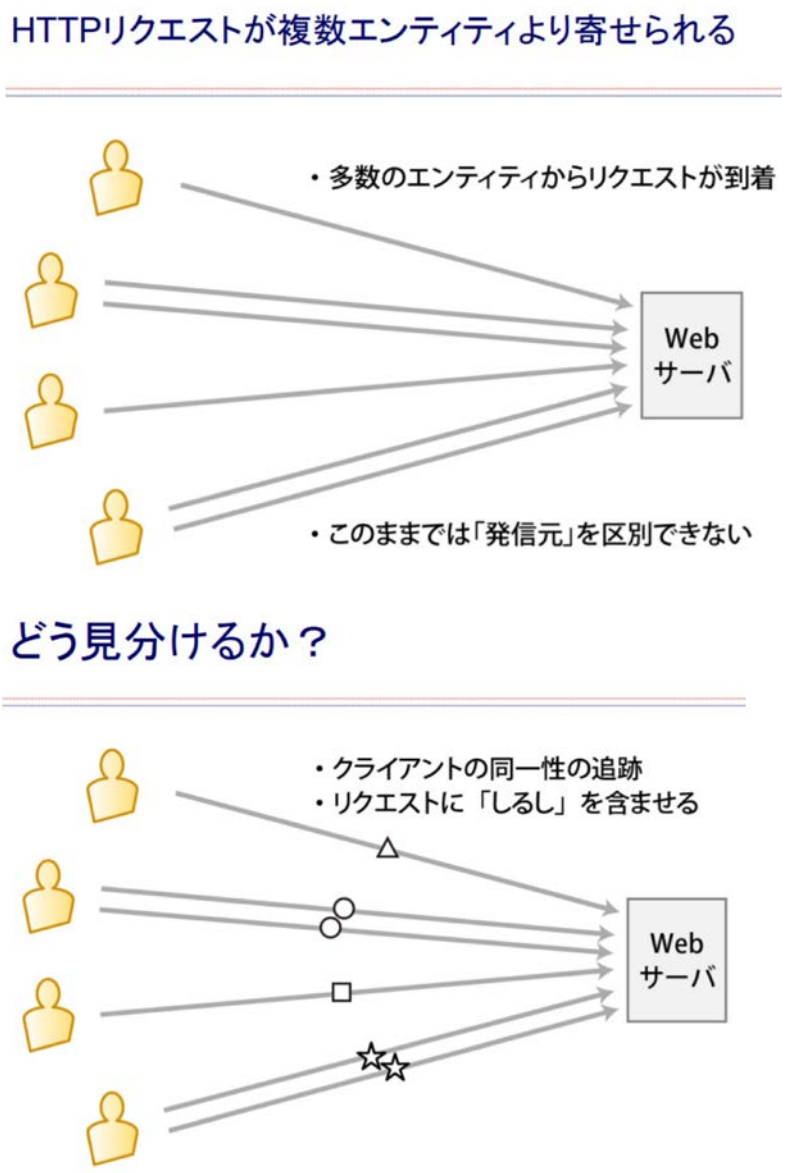


Web が登場した初期、Web ページは HTML で記述された静的なドキュメントのみだった。そのため、Web サーバの役割はクライアントからの HTML で記述された静的なリクエストに対してファイルサービスを提供することであり、ブラウザの役割は、サーバから提供されたファイル（HTML）を解析し情報コンテンツを表示することであった。したがって、サーバ上に現在の状態を表すデータなどを保持する必要がないステートレスな構造であったために、非力なマシンでもかなり多くの不特定多数に対してサービスを提供することが容易に可能となった。

その後、さまざまな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 など多くの言語を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クライアント・サーバ間のやり取りでの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ることになった。

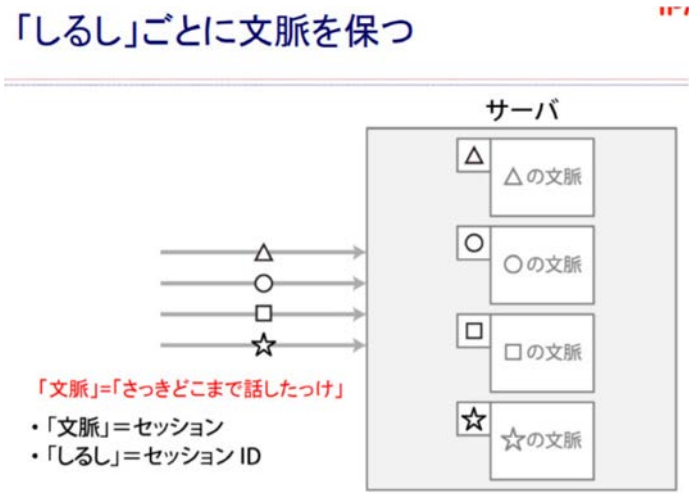
セッション管理とは、Web サーバに対する複数リクエストが同一クライアントから送信されたものであるかどうかを判断するために、クライアントとサーバ間で情報を保持し、アクセス制御を一つの集合体として管理する仕組みのことをいう。

図解 3-4　セッションの基礎とログインセッション



出所：独立行政法人情報処理推進機構（IPA）
「セキュア・プログラミング講座 Web アプリケーション編」
（2014 年 ブートアップセミナー）

図表 3-5　セッションの基礎とログインセッション　「しるし」ごとに文脈を保つ



出所：独立行政法人情報処理推進機構（IPA）
「セキュア・プログラミング講座 Web アプリケーション編」（2014 年 ブートアップセミナー）

始めは各々のアプリケーション自体でセッション管理機能の作り込みが行われ、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（https://www.ipa.go.jp/files/000030878.pdf）にも紹介されているので参照してほしい。

また、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いう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な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。この特性から生まれる代表的な脆弱性として以下のものがある。

・SQL インジェクション
・OS コマンド・インジェクション
・パス名パラメータの未チェック／ディレクトリ・トラバーサル

・セッション管理の不備

・クロスサイト・スクリプティング

・CSRF（クロスサイト・リクエスト・フォージェリ）

・HTTP ヘッダ・インジェクション

・メールヘッダ・インジェクション

・クリックジャッキング

・バッファオーバーフロー

・アクセス制御や認可制御の欠落

これらの脆弱性に対処する方法は、「安全なウェブサイトの作り方」(https://www.ipa.go.jp/files/000030878.pdf) が参考になる。また、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の次の資料を参照してほしい。

・Java セキュアコーディングセミナー資料
https://www.jpcert.or.jp/securecoding/

・Java セキュアコーディングスタンダード CERT/Oracle 版
https://www.jpcert.or.jp/java-rules/

2-2. C/C++言語の特性と注意すべき脆弱性

C 言語や C++言語は汎用性・移植性が高いことから、オープンソースソフトウェアや多くの製品ソフトウェアの記述に用いられている。コンパイラの入手が容易であることや、作られる実行可能ファイルがコンパクトであることも、多くの場面で採用される要因となっている。

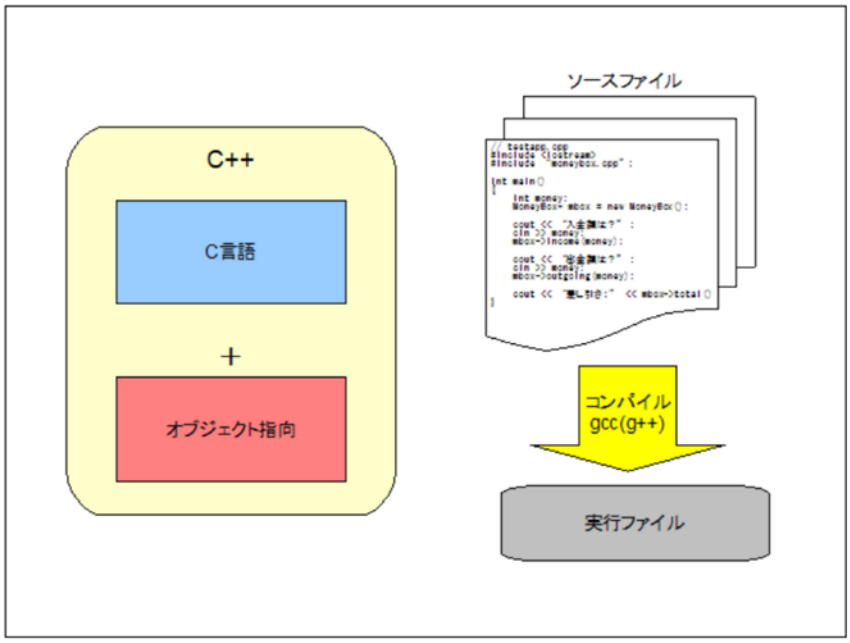
C 言語は、1972 年に AT&T ベル研究所によって開発された言語で、アセンブラとの親和性が高いことから、UNIX をはじめとして OS の記述言語としても広く用いられている。C 言語の精神として、プログラマを信頼し、プログラマがやりたいことができ

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。

このような特性から、C 言語はコンピュータを乗っ取るに至るようなセキュリティ脆弱性を生みやすい。代表的な脆弱性として、バッファオーバーフロー脆弱性、フォーマット文字列脆弱性、整数オーバーフロー脆弱性等があげられる。言語仕様の詳細を知らないプログラマが C 言語の特性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発生している。

一方、C++言語は C 言語が普及したのち、1980 年代に C 言語と同じくベル研究所で作られた。コンピュータシミュレーション記述言語から発展した Simula67 が持つ「クラス」の概念に基づくプログラミングを C 言語ベースで行えるよう、C 言語の拡張機能として考案されたものだ。C 言語で開発したプログラムを利用でき、他の言語よりも効率よく処理できることから急速に普及した。

図解 3-7　C++のプログラミング



出所：OSS 推進フォーラム http://ossforum.jp/node/941

C++はC言語の要素をほぼそのまま引継いでおり、新しい構文や機能を追加した形になっている。高度なオブジェクト指向プログラミングが可能である一方で、純粋なC言語のスタイルのプログラミングも行える。このことはC++言語でもC言語と同様、安全なロジックに欠けたプログラムがしばしば書かれ得ることを意味している。

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の次の資料を参照してほしい。

・C/C++ セキュアコーディングセミナー資料
https://www.jpcert.or.jp/research/materials.html
・CERT C セキュアコーディングスタンダード
https://www.jpcert.or.jp/sc-rules/

さらに、組込みソフトウェア開発向けについては独立行政法人情報処理推進機構（IPA）から提供されている次のものが参考になる。

・【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
https://www.ipa.go.jp/sec/reports/20140724.html
・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］
https://www.ipa.go.jp/sec/softwareengineering/reports/20130329_2.html

平成 30 年度「専修学校による地域産業中核的人材養成事業」
Society5.0 に対応した情報セキュリティ人材養成のモデルカリキュラム開発・実証事業

セキュアなネットワーク設計

平成 31 年 3 月

一般社団法人全国専門学校情報教育協会
〒164-0003　東京都中野区東中野 1-57-8　辻沢ビル 3F
電話：03-5332-5081　FAX　03-5332-5083