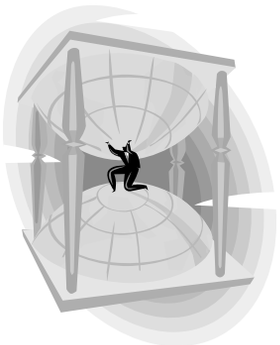
5 情報システムの脆弱性を意識したソフトウェア開発管理



森崎修司 * 久島広幸 **
（株）インターネットイニシアティブ技術本部企画開発部
*morisaki@iij.ad.jp **hisasima@iij.ad.jp

情報システムの脆弱性（vulnerability）は情報システムに存在する欠陥であり，正当な権限を持たない者に情報の入手，および，改変手段を提供してしまう．脆弱性が情報システムの存在自体をおびやかす可能性もあり，脆弱性対策の重要性は広く認識される必要がある．本稿では，典型的な脆弱性を示しながら，情報システムに含まれる脆弱性が与える影響を述べ，ソフトウェア開発の各フェーズにおいて脆弱性がどのように混入される可能性があるかを指摘し，対策方法を述べる．

脆弱性が与える影響と脆弱性対策の現状

脆弱性は20年以上前に文献1）で指摘されたものであり，現在RFC 789[2)]としても公開されている．脆弱性はセキュリティホールと呼ばれることもあり，その存在は比較的古くから知られている．しかしながら，現状，情報システムの開発者や利用者の脆弱性に対する意識は必ずしも高いとはいえない．重要な情報を扱う情報システムにおいても，既知の脆弱性が含まれていたり，脆弱性対策が十分でなかったりする場合もある．

社会において情報システムが重要な情報を扱う機会が増すとともに情報システムの脆弱性が与える影響は大きくなっている．電子政府や電子自治体，インターネットバンキングを始めとして，情報システムの持つ情報が重要になれば，脆弱性対策はその重要性を増す．たとえば，情報システムの脆弱性を利用し，正当な権限を持たない悪意のある者が，個人情報（たとえば，住民票，健康保険，銀行口座に関する情報）を入手したり，改変したりする可能性がある．

脆弱性対策はネットワーク事業者だけが留意すべきことではなく，情報システムを開発・利用するすべての組織が十分に検討すべきである．インターネットなどの公衆網を利用する情報システムは言うに及ばず，特定組織内に閉じている情報システムであっても脆弱性への対策は重要である．情報が重要になれば組織内であっても不正利用される可能性が大きくなるからである．不正利用が横行するとシステム廃止の可能性も考えられる．

文献3）でも述べられているように，現状，脆弱性対策はソフトウェアを出荷したり，利用し始めてから検討されることが多い．本来，脆弱性対策はソフトウェア開発の初期に検討され，開発中の各フェーズでも常に実施されるべきである．本稿では，ソフトウェア開発の各フェーズにおいて混入する可能性のある脆弱性について述べ，その対策について述べる．まず，ソフトウェアに含まれる脆弱性について具体例を挙げながら概説する．次にソフトウェア開発のフェーズごとに混入する可能性のある脆弱性について述べる．最後に現実的な対策について述べる．脆弱性が広義の意味で用いられる場合，パスワードの盗用，設定ミス，通信回線の盗聴等が含まれることがあるが，本稿ではこれらの脆弱性（運用面や組織，体制面で対策が可能な脆弱性）については対象としない．

脆弱性の具体例

◆推測可能な識別子

webブラウザを使用したオンラインショッピングサイトや社内ポータルサイトは身近なものとなっている．これらのサイトでは，webサーバ上のプログラムがユーザごとに異なるページを生成している．たとえば，“○○さんこんにちは”といったようなユーザ名を表示するページである．

ユーザごとに異なるページを生成する場合，ユーザを識別するための情報をリクエストごとにwebサーバ上のプログラムに送る必要がある．通常，これらのユーザ識別情報はURLの一部に含めたり（HTTPのGETメソ

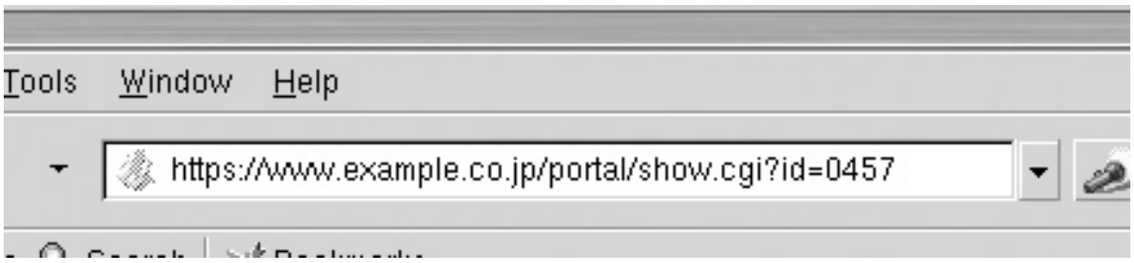


図 -1　ユーザ識別情報が含まれる URL の例

ッド），URL とは別に web ブラウザから送信する（HTTP の POST メソッド）．

ユーザ識別情報は web ブラウザやサーバプログラムだけが生成できるものではないため，正当な権限を持たないユーザでも識別情報を偽ることができる．そのため，与えられた識別情報が正当な権限を持つユーザのものかを判断せずそのまま使用する場合には脆弱性となり得る．たとえば，URL にそのままユーザ識別情報が含まれている場合には，URL のユーザ識別情報の部分を変更するだけで識別情報を偽ることができる．この脆弱性を除去するためには，毎回ユーザ認証を実施する，正当な権限を持つユーザしか知り得ない情報を併せて URL の一部に含める，等が必要である．

図 -1 は web サーバやデータベースに保管された個人情報を，web ブラウザを通じて閲覧できるシステムで使用する URL の一例である．web サーバ上で動くプログラムは URL の“?”以降でユーザを識別している．この例では，www.example.co.jp のサイトで提供される show.cgi というプログラムにパラメータとして 0457 を渡して実行している．web サーバ上で実行される show.cgi プログラムはユーザ ID が 0457 であるユーザ向けの情報を表示する．もしもユーザが URL に含まれる ?id=0457 の部分を ?id=0458 に変更するだけで，ユーザ 0458 のユーザの情報を閲覧できる場合には，これが脆弱性となる．

◆バッファオーバフローによる制御奪取

特殊な入力を与え，関数やサブルーチンの復帰アドレスが格納されるメモリ領域（バッファ）を不正に書き換えることにより，プログラムの制御を奪う方法の 1 つがバッファオーバフローによるものである．プログラムの制御を奪えば結果として対象プログラムのユーザ権限を奪うことができるため，一般にユーザ権限奪取と呼ばれる．実行時にメモリ境界のチェックをせず，メモリ領域に関数やサブルーチンの復帰アドレスを保持するような構造のプログラミング言語やそれを許す OS で稼働するシステムに，この脆弱性が含まれる場合がある．

C，C++ 言語はバッファオーバフローが起きる典型的なプログラミング言語である．C，C++ 言語では，関数の復帰アドレスは一部の変数（auto 変数）と共にスタック領域に配置される．変数のメモリ境界から溢れるような入力を注意深く選び，スタック上の復帰アドレスを上書きできれば，本来復帰するべきアドレスとは異なるアドレスにあるプログラムを実行できる．具体的には，入力に機械語コードを含めておき，復帰アドレスがその機械語コードの先頭を指すような値をプログラムに与え，元のプログラムの制御を奪う．

図 -2 は脆弱性を含む C 言語プログラムの例である．**図 -3** は図 -2 を実行した際のスタックの様子を示している．この図を用いてバッファオーバフローにより復帰アドレスが書き換えられる様子を述べる．通常，図 -2 のプログラムを実行すると，実行時引数 argv[1] に代入された文字列を src[128] にコピーし，それをさらに関数内の dst[64] にコピーし終了する．しかし，引数を注意深く選んで与えることにより，復帰アドレスを書き換え，関数 copy の終了後に任意のアドレスに制御を移すことが可能であり，実行時引数 argv[1] に機械語コードを含めておくと，プログラムから制御を奪うことができる．

図 -2 のプログラムの main 関数を実行すると，2 行目で図 -3（a）のように src[128] がスタック領域に確保される．3 行目では，argv[1] 引数で与えられた値を src へコピーする．4 行目で copy 関数を実行する際に，図 -3（b）のように関数 copy に渡される引数 *src，関数から復帰する際の復帰先アドレス，copy 関数のスタックフレームのベースアドレスがスタックに積み上げられる．9 行目で strcpy を実施すると src で指されるアドレスを先頭として，終端を示す NULL までの間にある値がコピーされる．その長さが 64 バイト以上の場合，図 -3 中のグレーの部分が示すように，スタックフレームのベースアドレス，関数からの復帰アドレス，copy 関数への引数の値が src に格納された値で上書きされ，さらに，src 自体の一部も上書きされる．

```
1:  int main(int argc, char *argv[]){
2:    char src[128];
3:    strncpy (src, argv[1], sizeof(src) -1 );
4:    copy(src);
5:  }
6:
7:  void copy(char* src){
8:    char dst[64];
9:    strcpy(dst, src);
10: }
```

図 -2　脆弱性を含むプログラムの例

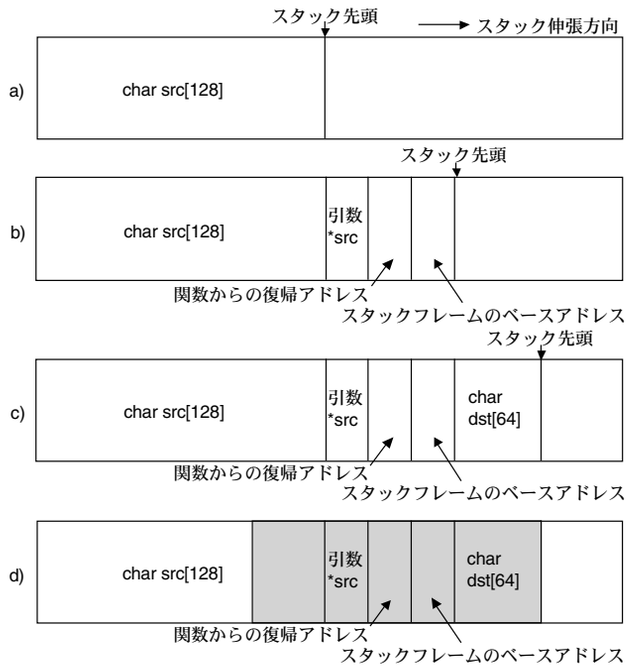


図 -3　プログラム実行に伴うスタックの変化

引数 argv[1] の一部に機械語コードを含めておき，復帰アドレスをその argv[1] の機械語コードの先頭とすれば argv[1] で指定したプログラムを実行することができる．たとえば，UNIX では execve（プログラムを起動するシステムコール）とシェル（/bin/sh 等）を組み合わせることで行うことができる．その際に，execve の引数として実行させたいスクリプトを指定すれば，シェルがそのスクリプトを実行する．このしくみを利用するために，引数を指定した上で execve を実行する機械語コードを argv[1] に含める．そうすることによって任意のスクリプトを実行できる．

図 -2 のプログラムでは，9 行目のコピーと 3 行目のコピーは一見同様のものであるが，3 行目の strncpy はスタック上の変数領域を上書きしないよう，コピーする文字列の長さを第 3 引数で渡して制限している．

実際には，実行する計算機の OS や CPU の命令セット，アーキテクチャ，エンディアン，サブルーチンコールのやり方等に依存するが，本質的には，復帰アドレスを書き換え，入力に含まれる命令を実行することによって制御を奪う．

◆ script injection

システムの入力としてスクリプト言語（シェルスクリプト，Ruby，Perl，JavaScript，SQL ステートメント等）のコードを与え，正当な権限なしにそのコードを実行しようとすることを script injection と呼ぶ．特に SQL ステートメントを入力として与えることによって，正当な権限なしで SQL ステートメントを実行しようとすることを SQL injection と呼ぶ．SQL injection に対する脆弱性は企業内システムや官公庁，大学内システムなどのデータベースに個人情報を格納し，従業員や職員，学生が自由に自分の情報を取り出せるようなシステムにも存在し得る．ユーザ認証時に入力するアカウント名やパスワードに対して SQL injection が可能であることを次のユーザ認証手順を用いて例示する．

1. ユーザは端末からキーボードを使って，ユーザ ID（社員番号や職員，学生番号等）とパスワードを入力する．
2. システムはユーザが入力したユーザ ID とパスワードがデータベースに格納されているものと一致しているかを SQL ステートメントで問い合わせる．

1 で入力されるユーザ ID は，通常，ユーザ ID，もしくは，その入力間違い程度であることがほとんどである．しかし，ユーザが SQL ステートメントを注意深く選んで入力し，それが SQL ステートメントとして実行できれば脆弱性となる．

図 -4 はユーザ認証に使用される SQL ステートメントの例を示したものであり，図中のグレーの部分がユーザの入力を表している．図 -4 の（a）はユーザが入力した ID “1234” を検索キーとしてユーザのパスワードを得るための SQL ステートメントである．図 -4 の（b）はユーザの入力に SQL ステートメントの終端記号である “;” が含まれた場合に，ステートメントが 2 つに分けて解釈，実行される様子を表している．前半のステートメントは，ユーザ ID に対応するパスワードを取得する部分であるが，後半はユーザ ID が 1235 のユーザのパスワードを aaa に変更するものである．

例は説明のため非常に簡略化されたものであり，これだけでは必ずしも他人のパスワードを変更できるわけではないが，このようなシステムに脆弱性が含まれる可能性を示しているといえる．個人番号としてユーザが入力した文字列が SQL ステートメントとして実行され

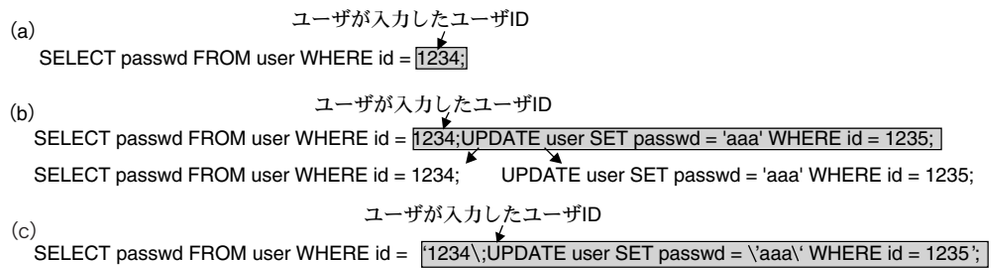


図 -4　ユーザ認証時の SQL injection の例

ないよう妥当性検査をすることにより，このような脆弱性を除去できる．具体的にはユーザが入力した文字列のうち，終端記号等の制御文字を通常の文字列として扱うようクォートする．図 -4（c）では“;”を SQL ステートメントの終端記号としてではなく，文字として解釈されるようバックスラッシュを挿入している．これにより，2 ステートメントに分割され解釈，実行されることはなくなる．

ソフトウェア開発フェーズで混入する脆弱性

ソフトウェアに脆弱性が混入する可能性は，ソフトウェア開発の大半のフェーズにおいて存在する．以降では各開発フェーズでどのような脆弱性が混入されるかを述べる．

◆要求分析フェーズ

通常，要求仕様には必要な機能に関する事柄しか書かれていないことが多いため，要求分析のフェーズで脆弱性が混入することはそれほど多くない．他システムとの互換性のために，特定のバージョンのライブラリやサブシステムを使用することが明記され，それらに脆弱性が含まれている場合には脆弱性の混入につながる場合がある．具体的には，脆弱性がすでに明らかになっているデータベースをサブシステムとして使用する場合がこれにあてはまる．

先にも述べたが，必要な機能がすべて正しく実現されている場合でも脆弱性は存在し得る．また，要求仕様に“脆弱性がないように”と記述することは可能ではあるが，現実には意味をなさない．ただし，要求仕様に“脆弱性混入防止のためユーザ入力の妥当性検査をする”というような脆弱性に関する具体的な記述を含めておくことは有効である．

◆設計フェーズ

設計フェーズにおいては，ユーザからの入力に対する妥当性評価や権限を持つ特定のユーザに許可を与えるとともに，権限を持たないユーザには暗黙的にも許可を与えないような設計とする必要がある．たとえば，前述の推測可能な識別子の例では，URL に含まれる ID を権限を持たないユーザが修正しただけでは，他のユーザの情報を閲覧できないような設計とする必要がある．また，ユーザの入力がプログラムとして実行できないような設計とする必要がある．

開発するソフトウェア以外でも多くの点に配慮する必要がある．具体的には，以下の項目が挙げられる．

- オペレーティングシステム
- ライブラリ
- コンポーネントソフトウェア
- サブシステムとして利用するアプリケーションソフトウェア

上述の項目に 1 つでも脆弱性が存在すれば，情報システム全体に対する脆弱性につながる可能性があるため，すべての項目について調査が必要となる．共同開発や委託開発においてサブシステムを異なる組織で開発する場合も同様に，それらサブシステムが調査の対象となる．

一般に古くから使われている（いわゆる枯れている）ソフトウェアだからといって，含まれている脆弱性が必ずしも少ないというわけではない．同一のソフトウェアでも予想されていなかった形態で利用するだけで脆弱性が露呈することも多い．

◆コーディングフェーズ

コーディングフェーズで混入される典型的な脆弱性はバッファオーバフローによるものである．C 言語やアセンブラを始めとして，変数が格納されるメモリ境界の実行時検査が必ずしも十分ではないプログラミング言語を使用する場合には，外部からの入力を格納する際にメモリ境界を検査することやメモリ境界を検査するライブラリを使用することが重要である．

脆弱性が含まれるのは開発するプログラムだけとは

限らない．インタプリタ，コンパイラ，ライブラリ等に脆弱性が含まれる場合もある．また，プログラムコードを自動生成するCASEツールを使用して実装する場合には，ツールが自動生成するプログラムコードに脆弱性が含まれる場合があることを認識しておく必要がある．

◆テストフェーズ

テストフェーズで脆弱性が混入する可能性はそれほど高くないが，すでに混入している脆弱性を除去できなかったり，バグ修正の際に新たな脆弱性を混入する場合がある．要求分析，設計フェーズで明示的に定義された脆弱性対策やコーディングフェーズの脆弱性対策が正しく実現できていることを確認できるようなテストを実施する必要がある．特に，script injectionに対するテストにおいては，ユーザのキーボードからの入力はもちろんのこと，ファイル名やメールアドレス等の間接的な入力についてもテストの対象としなければならない．

◆保守・運用フェーズ

日々新たな脆弱性の発見や新たな手法が現れるため，システムやソフトウェアに変更を加えない場合でも，脆弱性対策が必要になる．保守・運用フェーズにおいて脆弱性対策を怠ることは致命的となる可能性が高い．テストフェーズで除去できなかった脆弱性はもちろんのこと，新たに発見・報告される脆弱性についても対策が必要である．脆弱性は開発したソフトウェア，使用しているサブシステムやシステムを構成するハードウェアのファームウェアに至るまであらゆる所に含まれる可能性がある．

ソフトウェア開発における脆弱性対策

◆開発面での対策

・各フェーズのプロダクトのウォークスルー，インスペクション

仕様書，設計書，ソースコード等，各フェーズで生成されるプロダクトに対して脆弱性の有無を確認することを目的としたウォークスルー，インスペクションを実施することは効果的である．通常のバグと同様に，脆弱性も早期に発見できればできるほど修正にかかる工数は小さくなる．ウォークスルーやインスペクションの際に利用するチェックリストやシナリオの1つを脆弱性対策にあてると効果的である．具体的には，正当な権限を持つ者に対して提供される機能が，正当な権限を持たない者に対して与えられていないかを検査するためのチェックリストやシナリオを用意しウォークスルーやインスペクションを実施する．

・コーディング

Design by Contract[4]で提案されているように，仕様を満たしていない入力に対して警告を出力したり，状況によってはプログラムを停止するような仕組みを持たせたりすることも必要である．開発言語によっては，そのような機能を有しているものもあり（C言語のassertマクロ，Javaのassert構文等），コーディング中はそれらの機能を利用すると効率的である．

メモリ境界の実行時検査を行わないプログラミング言語を使用する場合には，変数領域を越えてデータを書き込む可能性の有無を検査する必要がある．使用するライブラリや関数によっては，上限を指定して領域を越えないように指定することができる．たとえば，C言語のライブラリでは文字列をコピーするstrcpy（）関数のかわりにstrncpy（）を使用し，コピー先の大きさを引数で指定することにより実現できる．

script injectionを防ぐためにはユーザの入力を制御文字や命令として実行しないよう，制御文字，命令をクォートして無効とする必要がある．使用するスクリプト言語のBNF（Baccus-Naur Form）等を参照しながら，クォートが十分であるかを確認すると効果的である．RubyやPerl言語では，ユーザからの入力をそのまま保持している変数に対して，妥当性を検査していないことを示すtaintedマークをつけることができ，マークがついている変数を使用する場合には，実行可能な命令が限定される．検査が十分かどうかを判断するのはあくまで実装者の判断に委ねられているが，このようなフレームワークを利用することは，特に，多人数でのコーディング時に役立つ．

・テスト

通常のテストに加えて，脆弱性の有無を調べるためのテストをする必要がある．仕様書の脆弱性対策に関する記述が十分であれば問題ないが，手薄な場合には改めてテストケースを洗い出す必要があるかもしれない．テストではすでに述べた仕様，設計に含まれる脆弱性，ならびに，バッファオーバフロー等の脆弱性がないことをチェックできるようなテストケースを分類しておき，前者から実施することが望ましい．後者はコーディング，テストフェーズでしか検出できないため，入念なチェックが必要となる．script injectionに対するテストでは，ファイル名やメールアドレス等の間接的に利用される入力によるテストが必要な場合がある．これらの入力は外界の制約を受けるため，システムの実稼働環境やテスト環境では，テストを実施できないことがある．その場合には，間接的な入力を生成するための専用の環境を用意し，テストを実施すべきである．外界の制約は新機能の追加やサブシステムやライブラリのバージョンアップ等によって，容易に変化するからである．

◆管理面での対策

・コストの明確化

システムが持つ情報のうち，守るべき情報とその脆弱性対策にかけてもよいコストを勘案し，明らかにしておくことが重要である．システムが稼働する組織において，すでにセキュリティポリシーが策定されていれば，それと照らし合わせながら決めていくことが効率的であり，望ましい．明確なセキュリティポリシーがない場合には先に作成するか，それに準ずるものを用意する必要がある．いずれの場合にも妥当性検証等を仕様の項目として含めておくことで，明示的に脆弱性対策へコストを配分できる．

・保守・管理体制

OS のバージョン，サブシステムとして使用しているソフトウェアのバージョン，使用しているライブラリのバージョン等をリポジトリで管理し，それらのいずれかが脆弱性を含むことが明らかになった場合に即座に対応できるようにしておくことが望ましい．特にサブシステムが組織の外部に公開されている場合や，脆弱性が公開され攻撃ツールが出回るようなものについては，迅速に対応できる体制を作っておくことが望ましい．

CERT[5] で毎日のように報告されているように，脆弱性は次々に発見されているため，脆弱性管理を継続的に実施できる体制を持つ必要がある．脆弱性管理は，OS, サブシステム，システムを構成するハードウェアのファームウェアに至るまで，システム全体にわたって実施する必要がある．インターネット上で公開される脆弱性に関する情報やベンダからの情報は常に注意しておくべきである．

・リスク分析

リスクとして考慮すべき項目として，脆弱性を利用され情報が漏洩した場合の影響範囲やそのインパクト，脆弱性が既知になってから修正用プログラムができるまでの期間，修正用プログラムが原因となる仕様変更等が挙げられる．システムによっては，脆弱性を防ぐことにより著しい性能低下を招く場合もあり，実行効率がきわめて重要であれば，性能評価のための期間を考慮しておくことも必要かもしれない．

◆外部組織・ツールの利用

・アウトソース

信頼のおける外部組織に開発や運用を委託するのも 1 つの選択肢である．外部組織とのやりとりの中で脆弱性が明らかになる場合もあるし，外部組織のノウハウを一部取り込むことができる場合もある．その際には信頼のおける外部組織を選定する必要があるが，一般には，脆弱性管理を継続して実施している実績があり，ネットワーク事業者を始めとして攻撃に対するノウハウを多く持つ組織を選ぶことが望ましい．

・ツールや外部組織による検査

他組織によるセキュリティ検査の実施やシステムの脆弱性検出ツールの使用を検討してもよい．ただし，通常，セキュリティ検査は既知の脆弱性が解決されているかどうかを調べることが主要な目的であること，脆弱性検出ツールは残念ながら現時点では成熟の域に達しているとは言いがたく，あくまでも補助的なものであることを認識しておくべきであり，それらの検査のみで脆弱性対策が完了したと考えるのは危険である．

脆弱性を意識したソフトウェア開発に向けて

脆弱性対策はソフトウェア開発時に十分に実施されていない場合が多く，運用開始後に発見されることが多いのが現状である．本来，脆弱性対策はソフトウェア開発時から十分に検討されるべきである．具体的には，正当な権限を持つ者のみが情報を入手，および，改変できる，ということのみならず，権限を持たない者は情報を入手できず，改変できない，ということが確実に実現できているか精査すべきである．精査の対象は開発するソフトウェアにとどまらず，OS, サブシステム，システムを構成するハードウェアのファームウェアに至るまで情報システム全体に及ぶ．また，日々新たな脆弱性が発見されている現状では，情報の入手・共有のための体制，ならびに，環境を用意し，脆弱性対策を継続的に実施できるようにする必要がある．

脆弱性に関する認識や対策はまだまだ十分であるとはいえず，いまだアドホックな方法でしか検討されていない．情報システムが社会にさらに広く浸透，発展するためには，まずは，脆弱性が与える影響を開発者だけでなく顧客やユーザが広く認識する必要がある．さらに，脆弱性を混入しないためのソフトウェアパターン，アンチパターン，テスト等の開発手法，ソフトウェア信頼度成長曲線（SRGM）等を用いた定量的な評価尺度の検討も必要となるであろう．

参考文献

1）Ghosh, A., Howell, C. and Whittaker, J.: Building Software Securely from The Ground Up, IEEE Software, Vol.19, No.1, pp.14-16（Feb. 2002）.
2）CERT Coordination Cener, http://www.cert.org/
3）Meyer, B.: Applying "Design by Contract", IEEE Computer, Vol.25, No.10, pp.40-51（Oct. 1992）.
4）Rosen, E.: Vulnerabilities of Network Control Protocols, ACM Software Engineering Notes, Vol.6, No.1, pp.6-8（Jan. 1981）.
5）Rosen, E.: Vulnerabilities of Network Control Protocols: An Example, http://www.ietf.org/rfc/rfc0789.txt

（平成 15 年 2 月 28 日受付）