セキュアプログラミングチェックリスト

1996 年 5 月 23 日

下記の文章は、分量を抑えたチェックリストです。ラボのエンジニアがセキュアな Unix コードを書く際にクイックリファレンスとして使えるようにしてあります。

この文章は、Simson Garfinkel と Gene Spafford による「UNIX & インターネットセキュリティ("Practical UNIX and Internet Security")」(O'Reilly & Associates, Sebastopol, CA, 1996, To order, call 1-800-998-9938) の 23 章から抜粋（あるいは脚色）されたものです。この本についてのより詳しい情報を下記の WWW の URL から得ることができます。:

http://www.ora.com/item/pus2.html（当時）
http://www.oreilly.com/catalog/puis/（現在）
http://www.oreilly.com/catalog/puis3/index.html?CMP=IL7015（第 3 版）
http://www.oreilly.co.jp/BOOK/puis/（日本語）

また、AusCERT の Danny Smith による論文"Enhancing Security of Unix Systems" からの情報も含まれています。このリストを編集してくださった Gene と Danny の両名に感謝します。

この文章の 1996 年の著作権は、O'Reilly & Associates 社および AUSCERT, The University of Queensland に帰属し、複製は、以下の事項が提供されたときのみ可能です。:

1) すべての複製が著作権表記およびクレジット情報全文を含む。
2) 複製について変更が加えられていないこと。
3) この文章は、著作権表記所有者の許可が表明されずに、いかなる他の文書、文集あるいは成果の一部として含めてはならない。

セキュアな Unix コードを書くためのラボのエンジニア向けチェックリスト

Rev.3C 5/23/96

セキュアな SUID プログラムとネットワークプログラムを書く

その答えは、「洗練すること」です。：Unix セキュリティモデルは基本的に堅実ですが、プログラマには注意が足りません。Unix における大部分のセキュリティ欠陥は、root もしくは他の特権で動作するプログラム中のバグや設計エラーに起因するか、あるいは、このようなプログラムの予期しなかった相互作用を通じて生じます。

セキュリティ関連バグを避けるための Tips

プログラムが特権ユーザとして動作したり、あるいは、何らかの意味合いで危険な状況で動作するとき、重要なのは、「そのプログラムを、できる限りバグフリーになるように作成すること」です。ここに、コーディングする際の一般的ルールのいくつかを掲げます。：

1. プログラムをすぐにコーディングせずに、慎重に設計すること。

2. すべての引数をチェックすること。ここでは、プログラム中の引数をチェックするとき、注意を払う必要がある具体的な箇所を掲げます。：
   * コマンドラインであなたのプログラムに渡される引数。
   * Unix システム関数に渡す引数。
   * 各変数の値について範囲チェックを行う。
   * データは、範囲を設けられる必要があり、文法上、問題がないかを検証し、できれば、データが生成されるところで検証する必要がある。
   * （愚かにも）環境から取得する引数。
   * ファイルから読みとる引数／入力。
   * 変数を意図する型に型変換（typecast）し、「入力が、その型として正しいか」を検証する。(例： signed int に渡すと負値になってしまう大きな整数を正値として使うことを望むとき。)

3．任意の長さの文字列を扱うとき、バッファの上限をチェックしない関数を使わないこと。

AVOID	USE INSTEAD
gets	fgets
strcpy	strncpy
strcat	strncat
sprintf	bcopy
scanf	bzero
sscanf	memcpy,
	memset

常に変数の長さの計算を正しく行うこと。すなわち、宛先の変数は、移動される変数の長さに対して十分に大きくなければならない。

決して、「文字列が正しく '' で終わっている」と想定してはいけません。すべてのデータ形態について制限チェックします。

決して「文字列は、最後に '' キャラクタをもつこと」を忘れてはいけません。それゆえ、"abcde" と言う文字列の長さは、6 バイトであり、（strlen 関数は 5 バイトと返しますが）5 バイトではありません！

4．システムコールからのすべての戻りコードをチェックすること。Unix オペレーティングシステムでは、ほとんどすべてのシステムコールが戻りコードを返します。

常に成功するとは想定しないこと。失敗によっては、（プロセス記述子、ファイル記述子、フリーなメモリ等の使い切りのように）ユーザが制御可能です。

5. Unix 環境変数に依存するようなプログラムをもたないようにすること。それゆえ、あなたが望むことは、下記のとおり。:

* 絶対に情報をプログラムの環境変数に渡さなければならない場合、プログラムが必要不可欠な環境変数をテストするようにし、その後、その環境を完全に消去します。まず、不要なものを消去し、必要なものを記入し、残りを無効にします。
* あるいは、単に、最も肝要な変数を残して、環境変数すべてをクリーンアップしてしまいます。
* 「意図するファイルが意図どおりにオープンされる」ようにし、「意図するファイルが意図どおりクローズされる」ように記述すること。すべての不必要なファイル記述子は、exec() を呼び出す前にクローズされる必要があります。
* 「あなたのシグナルが、理にかなった状態に設定されていること」を確認する。
* umask を適切に設定する。
* プログラムが起動するとき、明示的に適切なディレクトリに移動（chdir）する。
* あなたのプログラムが失敗してもコアファイルを残さないようにするため、あらゆる必要不可欠な極値を設定する。
* 理にかなった環境変数を exec() に渡す。（デフォルト環境ではない。）

6. 内部的な整合性のチェックを行うコードをもつようにすること。例えば、C でプログラミングする場合、その assert マクロを使います。

注意：

これは、製品コードにおいては良いことではありません。なぜなら、コアダンプを生成したり、かつ／または、（行数や assertion 表現のような）潜在的に取扱に注意を要する情報を漏洩したりする可能性があるからです。これは、デバッグコードのためのものであり、選択的に、有効にしたり無効にしたりできます。SIGABRT シグナルが捕らえられていない場合、コアダンプ（core dump）が生成される可能性があります。

assert マクロの概念は良く、そのプログラマは、作成目的について、クリーンアップし、整ったログ情報を報告し、きちんと終了する固有のものを設計することを望む可能性があります。

7. ログを多く記録するようにすること。ここに、記録することを望む可能性がある具体的な情報を掲げます。:

   * プログラムが実行された時刻
   * 実行したプロセスの UID および実効 UID（GID 情報も）
   * 実行した端末
   * プロセス番号（PID）
   * コマンドラインの引数。
   * 整合性チェックにおける不正な引数もしくは失敗
   * syslog の飽和問題を意識する。syslog に渡されるメッセージの総量を制限します。

8. あなたのプログラムの重要な部分をできる限り小さく、できる限りシンプルにすること。

9. あなたのコードを読み通すこと。「自分だったらどのように攻撃するか？」、「予期されていない出力を提供したら何が起きるか？」、「2 つのシステムコールの間で任意に、そのプログラムを遅延できたら何が起きるか？」について考えます。

10. あなたのプログラムがスーパーユーザとして何らかの機能を行う必要があるが、一般に SUID パーミッションは要求しない場合、その SUID 部分を別のプログラムにし、2 つのプログラムの間に注意深く制御・監視されるインターフェイスを作成することを検討します。

11. あなたが SUID もしくは SGID のパーミッションを必要とする場合、それらを意図した目的において合理的な限り、できるだけ早期に使い、次に、プログラムを起動したプロセスの実効 UID および実効 GID を返すことによって、それらを失効させます。

12. 絶対に SUID として動作しなければならないプログラムをもつ場合、汎用プログラミングインターフェイスをもつプログラムを採用することを避けることを試みます。

13. あなたのプログラムが特別なパーミッションを必要とする場合（例えば、高得点を特定のディレクトリに書き込むゲームプログラム）、root 以外の通常のユーザアカウントに SUID するようにすることを検討します。

14. 同じ目的を、新規グループに対して SUID することによって達成できる場合、何かを root に SUID しないようにすること。

15. あらゆる引数について、コマンドとファイルの両方について、常に完全なパス名を使うこと。

16. ユーザによってもたらされるあらゆるデータは、渡されて、ファイルに書き込まれたり、ファイル名として使われたりしますが、シェルのメタキャラクタについてチェックされる必要があります。
* ../ のようなもののみをチェックするようにしないこと。なぜなら、シェルによって処理された場合、これは、"./ のようなトリックを使うことによって失敗させられる可能性があるからです。
* 「有効な（valid）」キャラクタをチェックし、残りを捨てることが望ましい。

17. 運用環境を想定して、あなたのコードを吟味し、注意深くテストする。
例：
* 「プログラムが常に root 以外の者によって動作させられる」と想定する場合、そのプログラムが root によって動作させられたら何が起きるか？
* 「root によって動作させられる」と想定する場合、root として動作させられなかったら何が起きるか？
* daemon もしくは bin として動作させられるように設計された多くのプログラムは、root として実行するとき、セキュリティ問題をもたらす可能性があります。
* 「常に /tmp ディレクトリ中で動作させられる」と想定する場合、別のところで動作させられたら何が起きるか？

18. 入手可能なツールをうまく利用すること。あなたが C を使っており、利用可能な ANSI コンパイラーがある場合、それを使い、呼び出しについてのプロトタイプを使います。あなたが ANSI C コンパイラーをもっていない場合、よくある間違いをチェックするために、その lint プログラムを使うようにします。

19. あなたのプログラムをよくテストします。あなたが SVR4 に基づくシステムをもっている場合、(少なくとも) tcov というステートメントカバレージテスターを使うことを検討します。CodeCenter や Purify のような商用製品を使うことを検討します。(個人的な経験からは、「これらのプログラムは、非常に有用である」といえます。) University of Illinois の Brian Marick によって開発された GCT というテストツールも観てみましょう。「得体の知れないシステム攻撃者がバグを発見してくれるよりも、テストにおいてバグを発見する方がが良いこと」を銘記してください。

20. 競合状態について意識すること。これは、デッドロック、もしくは、2 つのコードの連続実行が失敗することとして、問題が明らかになる可能性があります。

デッドロック:「あなたのプログラムの複数のコピーが同時に動作している可能性があること」を覚えておいてください。あなたが操作するあらゆるファイルをロックすることを検討します。プログラムがクラッシュしながらも、ファイルがロックされたままとなるイベントにおいて、ロックを解放する手段を提供します。あるプログラムが ファイル A をロックして、次に、ファイル B をロックしようとしている一方で、別のプログラムが既にファイル B をロックしており、次にファイル A をロックしようとしているときに起きる可能性があるデッドロック等を避けます。

連続実行:「あなたのプログラムは、自動的に実行しないこと」を意識すること。すなわち、この問題は、あらゆる 2 つの操作の間に割り込まれ、別のプログラム (悪用を試みるプログラムを含む) が動作する可能性があるというものです。それゆえ、あなたのコードに、それらの間で任意のコードが実行された場合に失敗する可能性がある操作のペアがないか、注意深くチェックします。"stat"... "open"と"open"..."chown"のペアは、特に、悪名高いものです。

* プログラム環境は、たとえ (上記のように) ひとつのインストラクションでも、決して脆弱な状態のまま放置してはいけません。

21. あなたのプログラムがコアをダンプしないようにすること。コアをダンプする代わりに、あなたのプログラムが適切な問題を記録し、抜けるようにします。

22. シェルエスケープ (shell escape) を使わないこと。

23. 決して system() もしくは popen() 関数を使わないこと。execlp および execvp も疑われる。

24. open 関数で新規ファイルを作成することを期待している場合、関数を失敗させるために、ファイルが存在すれば、O_EXCL|O_CREAT フラグを使います。そのファイルがそこに在ることを期待する場合、その関数が失敗するように O_CREAT フラグを含まないようにします。

25. あるファイルについて、そのオーナーを変更したり、ファイルの状態を診たり、あるいは、そのモードを変更したりするような一連の操作を行うとき、まず、ファイルをオープンし、次に、fchown()、fstat()、もしくは、fchmod() のシステムコールを使います。これによって、あなたのプログラムが動作している間にファイルが置き換えられること（可能性ある競合状態）を防ぎます。

26. 「ファイルがファイルである必要がある」と考える場合、これがリンクではないことを確保するために、lstat() を使うこと。しかし、「ファイルが公衆のディレクトリ（cf#20., above）にある場合、あなたがそれをオープンする前にチェックできる事項は変化する可能性があること」を覚えておいてください。

    既存のファイルをオープンするために：

    - パスについて lstat() し、lstat が成功したことをチェックする。
    - それが受容可能であることをチェックする。（例： シンボリックリンクでないこと）
    - open() し（O_CREAT 無し）、open が成功したことをチェックする。
    - open によって返された fd について fstat() する。
    - lstat と fstat st_ino と st_dev フィールドが一致する場合、受容する。

    まだ存在しない新規ファイルを作成するために：

    - パスについて lstat() し、ENOENT を得たことをチェックする。
    - open(...,...|O_CREAT|O_EXCL,...) し、それが成功したことをチェックする。

あなたが本当に心配性である場合:

- open によって返された fd を fstat() する。
- 再度、パスを lstat() し、(a) それが存在することと、(b) それがシンボリックリンクでないことをチェックする。
- fstat と最後に返された lstat が st_dev および st_ino fields と一致することをチェックする。

注意：後者は、シンボリックリンクの試みに従わない O_CREAT|O_EXCL セマンティックに依存する。

Item Subject: Sec 2

27. テンポラリファイルを作成する必要がある場合、tmpfile() 関数を使うことを検討すること。これは、テンポラリファイルを作成し、それをオープンし、そのファイルを削除し、ファイルハンドルを返します。

28. エラー復旧（recovery）について。特権プログラムは、決して「特権とされたステータスによって、運用が成功する」とは想定できません。復旧が許可されていない限り、復旧を試みてはいけません。

29. 「/etc/utmp ファイル（これは、システムによっては書き込み可能）は、細工することによって攻略され、システム上のあらゆるファイルが書き換えられ、それゆえ、特権アクセスを得る可能性があること」を意識すること。utmp からの情報を使わなければならない場合、使う前にこれが正しいことを検証します。（例： tty に書き込むことを期待する場合、そのファイルは、/dev/ttyxx のようである必要があります。）

30. ダイナミック（動的）にリンクされたライブラリについて。SUID プログラムが特権で動作したまま非 SUID プログラムを呼び出す場合、システムライブラリをユーザが作成したライブラリと置き換えることが可能です。

31. chroot さえた場所以外に在り、プログラムによって要求されるファイルは、chroot() を呼び出す前にアクセスされなければなりません。

SUID/SGID プログラムを書く

1. 「SUID/SGID しないこと。たいていの場合、これは必要不可欠ではありません。」

2. 決して、SUID されたシェルスクリプトを書いてはいけません。

3. 特別なファイル群にアクセスするために SUID を使おうとしてはいけません。その代わりに、そのファイル群のための特別なグループを登録し、そのプログラムをそのグループに対して SGID します。SUID を使わなければならない場合、その目的のための特別なユーザを登録します。

4. 問題なく可能である場合、実行環境を消去し、改めて起動すること。多くのセキュリティ問題は、「攻撃者によってプログラムが動作させられた環境とプログラムが開発された環境の間に顕著な相違があったこと」によってもたらされてきました。プロセスが動作する環境変数全体は、そのプログラムによって検証され、再設定されます。これは、環境変数に、HOME、PATH および IFS のような既知の値を設定すること、有効な umask 値を設定すること、および、すべての変数を初期化することを含む可能性があります。

（注意： あなたの UNIX のバージョンが同一名の環境変数において複数の値を許容する場合、各値について、単に、"getenv" および "putenv"を使うことでは不十分です。環境変数全体を見捨てて新たに始める必要があるか、あるいは、「環境変数中に注意を要する変数の複数の複製が無いこと」を確認するために複数回の"getenv" を行う必要があります。）

5. system() および popen() を使わないこと。あなたが perl を使っている場合、パイプを使わないように試みます。別のプロセスを起動する必要がある場合、"fork" を execve(2)、exec(3)、もしくは、execl(3) 関数のみで使い、それらを注意深く使います。(execlp(3) および execvp(3) 関数を使わない。)

6. シェルエスケープ（shell escape）を提供しないこと。これらを提供しなければならない場合、ユーザのコマンドを実行する前に setgid(getgid()) し、setuid(getuid()) するようにしてください。追加的な処理のために留まることを期待する場合、最初に fork する必要がある可能性があります。

7. 一般に、あなたのコードを「(privilege bracket)」するために、setuid() および setgid() 関数を使います。setuid() かつ／または setgid() 特権が必要なときにのみ有効にすること。

8. パイプもしくはサブシェルを使わなければならない場合、環境変数 PATH および IFS については特に注意すること。問題なく可能である場合、これらの環境変数を消去し、これらにセキュアな値を設定します。
   例：
   putenv ("PATH=/bin:/usr/bin:/usr/ucb");

   注意：PATH 指定は、/bin:/usr/bin:/usr/ucb: （":"もしくは"::"のような空のパス名の連結）ではないことを確認してください。なぜなら、これは、黙示的に "."を PATH に加えてしまうからです。

   putenv ("IFS= tr");
   4.の「注意：」を参照

9. オープンするすべてのファイルについて、完全なパス名を使うこと。カレントディレクトリについて、いかなる想定もしないこと。(あなたのプログラム中の最初の段階の処理として、移動（chdir("/tmp")）することによって、この要件を強制することができます。)

10. root として動作させなければならないが、特定のディレクトリにあるファイルへのアクセスのみが必要である場合、まず、そこへ移動(chdir())する必要があり、次に、そのディレクトリを chroot() します。また、あなたが、例えば、/dev/null、/dev/zero、/dev/log のような他のファイルを必要とする場合、注意する必要があります。

11. SUID されたプログラムやスクリプトに taintperl を使うことを検討すること。

12. プロセスが動作する環境全体がプログラムによって、検証され、再設定される必要があります。これは、環境変数に HOME、PATH および IFS のような既知の値を設定すること、有効な umask 値を設定すること、および、すべての変数を初期化することを含む可能性があります。

パスワード：

1. ユーザが打鍵するに応じてパスワードを画面に返さないこと。

2. ユーザのパスワードをコンピュータ上に保管するとき、パスワードを暗号化すること。他に手段が無い場合、crypt(3) ライブラリ関数を使います。パスワードのソルト（salt）を選択する際には乱数を使います。そのユーザがパスワードを入力したとき、そのパスワードを同一のソルトで暗号化することによって、それが当初のパスワードであったか否かをチェックします。

3. シェルスクリプトから crypt(3) へのアクセスが必要な場合、ほぼ同様の機能を提供する /usr/lib/makekey を使うことを検討します。

乱数を生成する：

乱数については、豊富な知識が在ります。ここに、いくつかの一般的なルールを掲げます。:

* 乱数の場合、その数の各ビットは、0 と 1 が等確率である必要があります。
* 乱数の場合、その中の各 0 の後は、以降のビットにおいて 0 と 1 が等確率である必要があります。同様に、各 1 の後は、以降のビットにおいて 0 と 1 が等確率である必要があります。
* 数が大きな数のビットをもつ場合、約半分のビットが 0 である必要があり、他の半分のビットが 1 である必要があります。

セキュリティ関連目的のための乱数についての追加要件は、予測困難性です。:

* 乱数を生成するコンピュータについて、以前の出力、あるいは他の知識が与えられたとしても、乱数生成器の出力を予想することは不可能であること。
* 乱数生成器の初期状態を判定することは不可能であること。
* 乱数生成器の初期状態を再現すること、あるいは、同一の初期値で生成器にシードを渡すことは不可能であること。

乱雑なシードを選択する

1. 限られた空間から乱数生成器にシードを与える。
2. 乱雑なシードとして、現在時刻のみのハッシュ値を使う。
3. シードの値自体は公表する。
4. 暗号技術的に強いハッシュ関数を使って、早く継続的に変化する何らかのフィールドを組み合わせる。
   例：(netstat -a ; ps auxww ; vmstat ; who) | md5

システムによっては、この目的のために /dev/random デバイスを提供する可能性があります。

そこで、良い乱数をどのように選択しましょうか？ここに、いくつかのアイディアがあります。:

* 放射性の源泉、FM ダイヤル上の統計、熱雑音、もしくは、同様な真正な乱雑性の源泉を使うこと。
* ユーザに一定量のテキストを打鍵することを依頼し、打鍵間の時間をサンプルとする。
* ユーザを監視する。ユーザがキーボードを打鍵するたびに、今回の打鍵と前回の打鍵の間の時間を計り、それを現在の乱数シードに加え、暗号技術的なハッシュ関数でハッシュ結果を求めます。乱雑性をさらに高めるために、マウスの移動も使うことができます。
* システム時計に依存することを避ける。
* イーサーネットアドレスや、ハードウェアのシリアル番号は使わない。
* ネットワークパケットの到着時刻のような情報を使うことを意識する。
* 大きなデータベースからの無作為抽出を使わない。
* あなたのシステムに既に在るアナログ入力デバイスを使うことを検討する。

この文章の 1996 年の著作権は、O'Reilly & Associates 社、および AUSCERT, The University of Queensland に帰属し、複製は、以下の事項が提供されたときのみ可能です。:

1) すべての複製が著作権表記およびクレジット情報全文を含む。
2) 複製について変更が加えられていないこと。
3) この文章は、著作権表記所有者の許可が表明されずに、いかなる他の文書、文集あるいは成果の一部として含めてはならない。