# OSECPUの資料

セキュアなシステムを作ろうクラス
セキュアなOSを作ろうゼミ
講師　川合秀実

# 他のゼミ、他のクラスのみなさんへ #01

今年度のセキュアなOSを作ろうゼミでは、講師が中心となって
セキュアなOSを作り、その過程を見せつつ、自由に開発に参加してもらう
ことで、セキュアなOSを作ることについて具体的に学んでもらいます。

・講師が作るセキュアなOS開発プロジェクトを手伝うというテーマ
・自分で1からセキュアなOSを作っていくというテーマ
・その他の、このゼミにふさわしい関連開発テーマ

いずれのテーマでも、講師が作っているOSについての資料をまとめて
おけば参考になるだろうと考えて、このテキストを用意しています。

# OSECPUとは？ #01

基本
- セキュアなOSを自作するというプロジェクト
- 「おせくぷ」と読む
- オープンソース
- プロジェクトリーダー：川合秀実
- http://osecpu.osask.jp/wiki/
- 2012.09.09: Wiki立ち上げ
- 2013.03.19: 実装開始
- → つまりはまだ若いプロジェクト

JITコンパイル方式を採用
- セキュアにするため
- 特定のCPUに依存しない互換性
  （CPUが異なってもアプリはバイナリ互換）
- 十分に高速な実行速度

# OSECPUとは？ #02

機能密度

概念的な定義：機能密度 ＝ 機能の量 ÷ バイト数

OSECPUアプリの機能密度: 世界一

Windows上でOSECPUアプリを実行するためのVMのインストールサイズ：

30KB未満

→ つまりOSもアプリも機能密度が異常に高い

本当に機能密度が高いとはどういうことか

OSがほとんど何もせず、アプリに処理を押し付ければOSが小さくなるのは当然
→ その代わりアプリは大きくなる

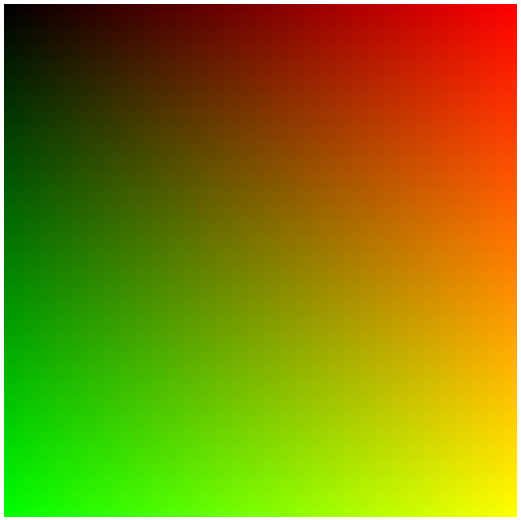
OSが何でも引き受けて、アプリに楽をさせれば、アプリが小さくなるのは当然
→ その代わりOSは大きくなる

両方が小さいのは本物
→ 設計がよい証拠（註：自画自賛です）

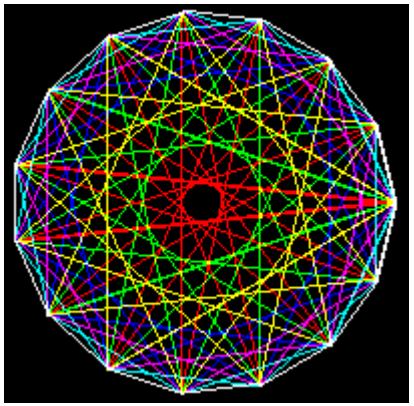
# OSECPUとは？ #03

ここまでのまとめ

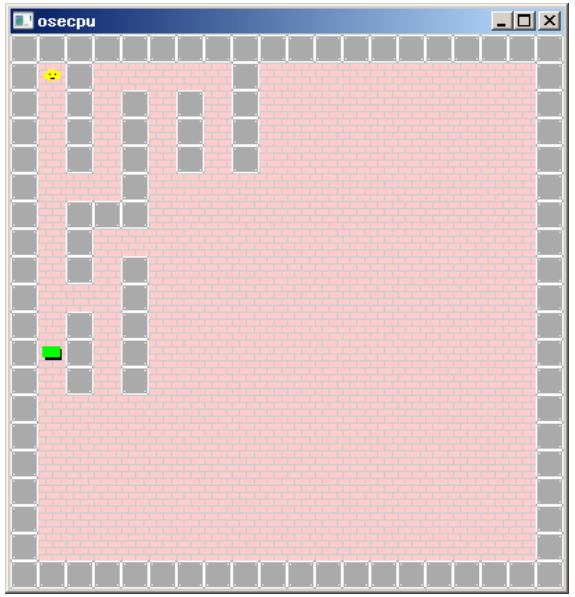
OSECPU ＝ セキュアなOS ＋ JITコンパイラ ＋ 機能密度技術



13バイトでグラデーション



71バイトで
こんな模様




284バイトでこんなゲーム

# OSECPUとは？ #04

OSのソースコードの規模（ver.0.64）
　osecpu.c　　3288行　- 下記以外のすべて
　tek.c　　670行　- 標準パッカー
　decoder.ask　1630行　- 内部コード変換
　→ ソースコードは（この手のものとしては）短いほう
　→ 逆に言うと、だからOSが30KBにも満たない

そのほか
　川合がセキュアなOSを作ろうと思ったのは、2011～2012年で
　セキュアなOSを作ろうクラスの講師をしていて、他の先生の講義を
　聞いているうちに自分でも作りたくなったため
　→ 講師すらやりたくなるほどの、それほど良いクラスだった
　　ということもでもある（笑）
　（註）この期間のクラス長は根津先生でした！（＝ 自画自賛ではないです）

# OSECPUにおけるセキュアの考え方 #01

基本
(1) 特定のCPUの特定の機能を使わないと実現できないような方法は
使わない
(2) 16bitや8bitのCPUもターゲットとする
(3) 単にシステムやデータを保護するだけではなく
デバッグ支援的なことも行う
(4) これらのせいで多少実行速度が落ちてもそれは気にしない

理由
(1) 幅広いCPUに対応できればアプリを移植しないで済む
→ 移植の際にバグってしまう危険性を排除できる
(2) これらのCPUを意識することで、内部設計がよくなる
(3)「脆弱性はバグの一種である」と川合は考える
だからデバッグ支援は脆弱性根絶の役に立つはず
(4) あらかじめ覚悟を決めておくことは重要
しかし(1)〜(3)と競合しない範囲ではもちろん速度も追及

# OSECPUにおけるセキュアの考え方 #02

考察(1)

最近のPC向けCPUは超優秀なので、速度低下はほとんど気に
ならなかった。これで移植不要でセキュアになるなら
悪くないバランスなのではないかと思う。

考察(2)

OSECPUはセキュアとJITコンパイラというありふれた要素のほかに
機能密度という珍しい要素を取り入れているが、これは良かったと思う。
単なるセキュアなOSだと、よほどのことがなければ、あまり関心を
持ってもらえなかったのではないかと思う。
→ つまり「何か自分の得意なプラスアルファをしたら、
うまく行きやすい」という例だと思う。
そういうのがないと、自分でも飽きてしまう可能性がある。

# OSECPUにおけるセキュアの考え方 #03

「セキュリティチェックをOFFにできる」という設計
　OSECPUはセキュリティチェックをOFFにできるように設計されている。

　たとえばx86も比較的強固な保護機構を持っている。だからセグメントを
　はみ出してしまったら例外を起こす。
　しかし、これらをOFFにできるという発想はない。

　OFFにできるというのはどういうことかというと、セキュリティチェックを
　利用して、アプリの動作の助けにするようなことは許さないということ。
　このポインタはどこまでアクセスを許してくれるのだろうか？ということを
　知るために、x86ではLSL（Load Segment Limit）という命令を用意して
　いるが、そんなものはOSECPUにはない。あってはいけない。

# OSECPUにおけるセキュアの考え方 #04

「セキュリティチェックをOFFにできる」という設計 [つづき]

なぜLSL命令はあってはいけないのか。それは、LSLの結果を正しく
生成するには、セキュリティ情報を常に生成しなければいけないから。
そうしなければ、ONとOFFとで結果が違ってしまい、アプリは
「セキュリティがOFFだったら悪さをする」というif文を組み込めてしまう。

セキュリティがOFFにできると何が嬉しいのか。そう、動作が高速になる。
もちろんセキュリティチェックがないので、悪意あるプログラムはやりたい
放題で危険だけど、これをユーザの自己責任で選択することができる。
そして仮にOFFにしても、アプリがそれを検知する方法はない（そんな
方法がもしあったら、それはOSECPUの脆弱性なので教えてください）。

# OSECPU-ASKA入門 #01

(0) はじめに

以下の記事はver.0.64のWindows版を前提にしています。
他の版を使っている場合は適宜読み替えてください。

→ たとえば osecpu064a.zip とかのことです

# OSECPU-ASKA入門 #02

(1) ASKAは意外に難しくない？

いきなりですが、以下のプログラムを見てください。

```
#include "osecpu_ask.h"
    junkApi_fillRect(4, 640, 480, 0, 0, 7);
    junkApi_fillOval(4, 300, 300, 320-150, 240-150, 1);
```

たったの3行です。中の数字の意味はとりあえず全く分かないでしょう。
それどころか英語の部分だってよく分かりません。とりあえずそのことは気にしないことにします。とにかく重要なことは、3行しか書いていないということです。
これをOSECPUの入ったフォルダにテキストファイルとして保存して
（ここではとりあえずapp0038.askとする）、Windowsのコマンドプロンプトで、

```
prompt>amake app0038
```

# OSECPU-ASKA入門 #03

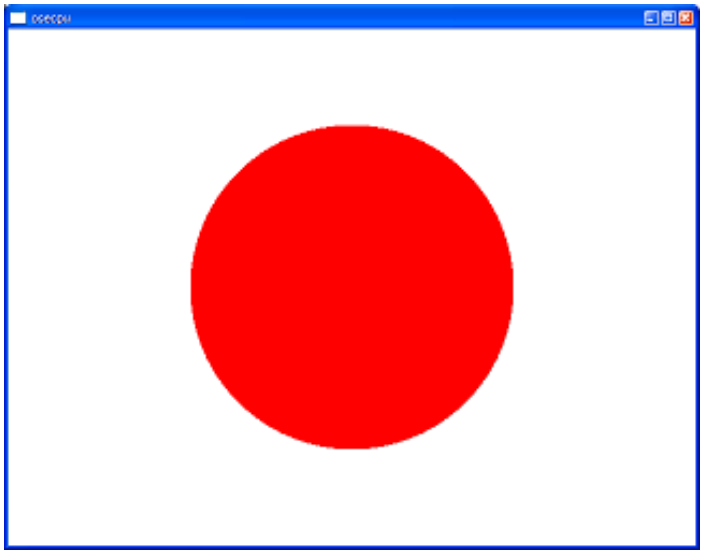
(1) ASKAは意外に難しくない？ [つづき]

と入力すると、app0038.oseというアプリケーションファイルができます（もともとあった場合には問答無用で上書きされます）。

次に

```
prompt>osecpu app0038.ose
```

とすればこれは直ちに実行されて、以下のような画面が見えると思います。



# OSECPU-ASKA入門 #04

(1) ASKAは意外に難しくない？ [つづき]

どうですか、つまり3行書くだけでこの程度の画像が表示できるというわけです。これは結構すごいと思うのですがどうでしょうか。CやC++でMFCとか使ってやったら、こんなに手軽にはできません。

（他の例もあるのですが、画面写真が単色印刷だと見づらくなりそうなので、続きはWebで http://osecpu.osask.jp/wiki/?page0036 ）

# OSECPUの内部仕様 #01

(1) 整数レジスタ

OSECPUは32bitの signed int な整数レジスタを64本持っています。
実装仕様としては64bit以上で演算する場合も許されているので、これは
最低でも精度が32bit分ある、ということになります。
→ したがって、0x7fffffffに1を足したら、値が負になることを保証している
わけではありません。
R00～R3Fと表記します。
R00やR01など番号の若いレジスタは、実際のCPUの実レジスタに
割り当てるように推奨されているので、R00への代入やR00の値の参照は、
たとえばR20に対する操作と比べて数倍高速になることが多いです。
ということで、特に必然性がないならR00やR01を使いましょう。

# OSECPUの内部仕様 #02

(1) 整数レジスタ [つづき]

これらのレジスタはすべて対等で汎用的に使われるというわけではなく、ある程度の使い方が決まっています。これに逆らってはいけないということはないですが、ライブラリなどで食い違うといろいろ面倒かもしれません。

R00～R1F（32本）: 最も汎用的な整数レジスタで、通常は関数ごとにローカルとして扱えます。つまり関数を呼び出しても破壊されたりはしません。

R20～R27（ 8本）: 汎用ですが、関数ごとにローカルというわけではなく、グローバル変数的に使うことを想定しています。関数呼び出しによって変更される可能性もあります。

R28～R2F（ 8本）: これも汎用ですが、関数ごとにローカルではなく、グローバル変数的に使われます。主にOSが用途を決定しています。これに対してR20～R27はアプリが自由に用途を決定できます。

# OSECPUの内部仕様 #03

(1) 整数レジスタ [つづき]

R30～R3B（12本）: 基本的にはこれらも汎用なのですが、関数の引数を渡したり、返値を入れたりするためにも使われるレジスタで、値が破壊されやすいです。ASKAでは複雑な数式を計算しなければいけなくなると、R3BやR3Aをテンポラリとして勝手に使い、値を破壊してしまうこともあります。R39やR38にまで手をつけることだってあります。しかし最悪でもR30までで、R2Fに手出しすることはありません。

R3C～R3E（ 3本）: 将来の拡張のためにリザーブされています。

R3F（ 1本）: 特別な用途のための整数レジスタです。汎用には使えません。

全体として、OSECPUのレジスタはかなり多いほうだと思います。これはOSECPUがメモリ操作を苦手としていて、できるだけレジスタだけで主要な演算が完結できるようにという設計方針によるものです。

# OSECPUの内部仕様 #04

(2) フラグレジスタはない

x86でもARMでも、さらには6502やZ80でさえも、みんなフラグレジスタというものを持っていました。

しかしOSECPUにはフラグレジスタはありません。MIPSの仕様に似ています。

フラグレジスタがない代わりにCMPcc命令の結果に応じて任意の整数レジスタを0か-1に変更することができます。つまり普通のレジスタをフラグレジスタの代わりにしてしまったようなものです。

これで設定された値をCNDプリフィクス命令（04 Rxx）で使えば、条件分岐や条件付き代入などができます。

# OSECPUの内部仕様 #05

(3) ポインタレジスタ

OSECPUはメモリアドレスを指し示すためのポインタレジスタを64本持っています。
P00～P3F と表記します。P00やP01がP20よりも高速に利用できます。
その他もRxxレジスタとほぼ同様で、用途が決まっています。

P01～P1F, P20～P27, P28～P2F, P30～P3B, P3C～P3E, P3F

ただし、P00レジスタは現在のバージョンでは使用できません。

ポインタレジスタは型情報を記憶していて、それと一致しないメモリアクセスを実行しようとすれば、セキュリティ例外が起きます。
ポインタレジスタはアクセス可能域情報を持っていて、それをはみ出してメモリアクセスしようとすれば、やはりセキュリティ例外が起きます。
ポインタレジスタはアクセス先のメモリの死活追跡情報も持っていて、freeしたメモリをアクセスした場合は、セキュリティ例外が起きます。freeしたあとで、たまたまそのメモリ域をmallocできた場合でも、ちゃんとセキュリティ例外が起きます。freeしたメモリをもう一度freeしようとしたときも、セキュリティ例外になります。多くのやっかいなバグや脆弱性を未然に防げます。

# OSECPUの内部仕様 #06

(4) セキュリティ例外

OSECPUはセキュリティ例外を起こすと、ソースコード上での行番号を表示して、
実行を停止することができます。巨大なコアファイルを出力して強制終了する
わけではありません。
停止ではありますが、原則として再開はできません。
停止なので、画面状態などは、例外を起こした瞬間のまま止まっています。
じっくり観察し、原因を確認することができます。
簡易モニタプログラムが起動するので、任意のレジスタの値を確認することも
できます。ポインタレジスタの中身も詳細に確認できます。
将来的にはメモリの値やmalloc/free履歴なども確認できるようにする予定です。

# OSECPUの内部仕様 #07

(5) JITCというAPI

OSECPUにはJITCというAPIがあります。これはJITコンパイルのことです。
このAPIは任意のバイト列の配列を渡すことができます。システムはこのバイト列を直ちにJITコンパイルし、その先頭アドレスをポインタレジスタに返します。
もちろんこのポインタレジスタの先へジャンプすることが可能です。後付けで関数を作るAPIだと考えたらわかりやすいかもしれません。
つまり、アプリはバイトコードのevalができるということです。これはx86などの実CPUでは当たり前すぎる機能ですが、Javaや.NETでは標準的にはサポートされない機能です。
レジスタもメモリ空間も基本的には共有します。したがって、ポインタレジスタを引数に渡してやれば、問題なくデータを受渡しできます。関数へのポインタを渡せば、コールバックもできます。

# OSECPUの内部仕様 #08

(5) JITCというAPI [つづき]

このAPIはセキュリティという観点では地雷に等しいのですが、しかしなんとか手なずけています。むしろこの機能が安全に提供されることで、OSECPUには多大な可能性があります。

たとえばOSECPU上で独自のJITコンパイラを機種依存なく構築することができます。普通、JITコンパイラを作るとなったら、x86用とか、ARM用などと、ターゲットを決めて、そのアーキテクチャ用の機械語を生成しなければいけません。複数のアーキテクチャに対応させるとしたら、そのすべてに対応する必要があります。これはJITコンパイラ作者にとっては負担です。しかしOSECPUならOSECPUのバイトコードさえ生成できれば、あとはOSECPUが面倒を見てくれます。しかもOSECPUなので、その独自JITコンパイラも容易にセキュアにできます。

# OSECPUの内部仕様 #09

(5) JITCというAPI [つづき]

他の例を挙げます。現在設定ファイルというと value = 1 のようなものをテキストで記述した *.ini というファイルをよく見かけますが、これらはそれぞれのアプリが独自にパーサを持ち、解釈しています。これは非常に無駄ですし、バグなどがあれば脆弱性にもなりかねません。おかげで仕様もアプリごとにまちまちです（特にコメントの書き方とか）。

こんなものはやめて、設定ファイルをOSECPU-ASKAのソースにしたらどうでしょうか。そうすれば設定ファイル中で、単なる代入だけではなく、加減乗除や条件分岐、一時変数の利用やループまで記述できます。

これをコンパイルしてバイトコードにし、それを設定ファイルにするのです。

アプリはこの設定ファイルを読み込んで「実行」するのです。そうすれば設定値が変数にセットされた状態で帰ってきます。何もパースする必要はありません。

# OSECPUの内部仕様 #10

(5) JITCというAPI [つづき]

設定ファイルが勝手に画面に落書きをするんじゃないかとか、そういう心配があるかもしれませんが、設定ファイルからはAPIを使えなくすることは可能ですし、特定のAPIだけ使えるようにすることも簡単です。
設定ファイルが無限ループに落ちてしまうんじゃないかとか、メモリリークするんじゃないかとか、そういう心配も（OSECPUの将来のバージョンなら）無用です。
そういう事態になったら実害が起きる前に設定ファイルを停止させて、セキュリティ例外とし、親アプリに戻ってエラー報告して実行を再開します。アプリ側は、設定ファイルの確保したリソースなどを処分してもいいですし、ユーザにエラー表示することもできます。

これらの機能をたかだか30KBのプログラムが提供するのです。まあ将来的にはもう少し大きくなるかもしれませんが、それでも100KBになることはないでしょう。しかもこの程度のものを個人が数か月で作れる時代なのです。他人の作ったOSやVMに文句を言っている場合ではないと思いませんか？

# OSECPUの内部仕様 #11

(5) JITCというAPI [つづき]

OSECPUは将来的にはシェルなども実装し、これによって機種依存、環境依存なく同じシェルが使えるようになるのですが、しかしそのシェルは、独自のシェルスクリプトを提供しません。それはまさに設定ファイルの時と同様に、JITCのAPIで実行すればいいからです。つまりアプリとシェルスクリプトの差異はほとんどありません。

こうすることで、ユーザは独自の雑多な言語仕様を受け入れる必要が無くなります。ユーザは自分の好きな言語でOSECPUのバイトコードを作ればいいのです。

現在、OSECPU向けのプログラミング言語を設計している人が何人もいます。どんどん可能性は広がっているのです。

# OSECPUのセキュアの仕組み #01

(1) 整数レジスタはチェックしない

OSECPUでは整数レジスタの値について、それが適正な値なのかどうかというチェックをAPI以外ではしていません。
もちろんこれを全面的にやったほうがより安全で、バグが見つけやすくなるということは分かります。しかし以下の点でこれをやらないことに決めました。

・値を間違えても、致命的なことにはならない。
　→ APIなど、致命的になるときには必ずチェックしています。
・ライブラリやAPIでチェックしていれば、比較的早期に発見できる。
・これをこまめにチェックすると実行速度がかなり落ちる。

# OSECPUのセキュアの仕組み #02

(2) ポインタレジスタに対するチェック

ポインタレジスタはメモリアクセス時にたくさんのチェックを受けます。
なぜなら不正なポインタを見逃してしまうと追跡困難なバグや脆弱性が次々と発生してしまうからです。

・配列の場合、配列の外に出ていないかどうか

・型は一致しているか

→ SInt32のメモリにUInt16で読み書きすることは許さない

→ この制約のおかけで必ず型が一致するので、エンディアンが異なる環境でも結果が同一になる

→ プログラムコードへのポインタとデータへのポインタは型が違うのでエラーになり、データへ分岐するということはできない

・そのメモリ域はまだ生きているか

→ すでにfreeされていることを、OSECPUでは「死んでいる」という

# OSECPUのセキュアの仕組み #03

(3) チェック情報はどこにあるか

ポインタレジスタは256ビットの構造体になっています。
純粋なポインタはその中の32ビットだけで、残りはすべてセキュリティチェックのための情報です。
・配列の場合のアクセス可能域、型情報、
死活管理構造体へのポインタ、死活チェックリビジョン番号

ポインタレジスタをメモリに書き込めば、これらの情報も一緒に書き込まれます。ポインタレジスタを他のレジスタにコピーすれば、これらの情報もすべてコピーされます。

チェック情報を変更することは原則としてできません。ただし配列の場合にアクセス可能域を狭めるための命令はあります。

# OSECPUのセキュアの仕組み #04

(4) 死活管理情報

ここには指定されたメモリ域が、いつどこのメモリアロケート命令によって確保されたものなのか、メモリ域の大きさは何か、型は何か、などの情報が登録されます。そしてリビジョン番号もあります。
もしこの管理先のメモリがfreeされると、リビジョン番号が1増えます。
そしてこの管理情報は、やがて他のメモリ域の情報を管理することになります。

このリビジョン番号がなければ、管理情報の使いまわしができなくなるので、malloc-freeをたくさん使うプログラムに対応できなくなってしまいます。

ポインタレジスタでメモリアクセスする際には、レジスタの中のリビジョンとこの管理情報のリビジョンとを比較します。一致しなければエラーです。

# OSECPUのセキュアの仕組み #05

(5) 不正な分岐への対策

たとえばx86では、コールゲートという仕組みがあって、アプリから
システムへの意図しないアドレスへの分岐をブロックしています。

OSECPUにはコールゲート的な仕組みはありません。システムコール
（＝API呼び出しもこれに該当）は、ただのCALL命令です。
API呼び出しを例にとれば、アプリは起動時にP28というポインタレジスタを
受け取ります。このアドレスをCALLすればいつでもAPIが利用できます。

このアドレスに適当な整数を加えてCALLしたらどうなるでしょう。もしこれが
成功すれば、システム内の任意の関数を呼び出せてしまいます。しかし、
コードを指すポインタレジスタは、加算や減算でポインタをずらすと
呼び出し不能なポインタになってしまうので、この攻撃は成功しません。

# OSECPUのセキュアの仕組み #06

(5) 不正な分岐への対策 [つづき]

OSECPUでは、アプリはアプリの中の任意のコードラベルに自由に分岐できます。しかし、アプリの外のラベルには分岐できません。分岐したければ、何らかの方法で外部へのポインタを取得しなければいけません。つまり、ポインタをもらう手段がなければ、どうやっても外部に手出しはできません。

「OSECPUの内部仕様 #10」で、設定ファイルにAPIを使わせない方法があると書きましたが、要するにP28をNULLにしてから呼び出せばいいのです。こうなると設定ファイルはAPIを呼び出せません。また、何かダミーの関数を用意して、そのアドレスをP28に設定し、それから設定ファイルを呼び出せば、設定ファイルがAPIを呼び出してきたときに「検閲」できます。許したいAPI呼び出しであれば、ダミー関数が本物のAPIを呼べばいいですし（代行）、許さない呼び出しであればブロックすればいいでしょう。

# OSECPUのセキュアの仕組み #07

(6) 無限ループ対策

OSECPUでは、無限ループや極端に重い処理への対策があります。
というのはこれがないと、設定ファイルをCALLしたとたんに帰ってこない、
という事態が生じるかもしれないからです。
OSECPUでは、分岐命令を実行するたびに、分岐回数カウンタという
内部変数が+1されます。また、分岐しない命令が16回程度連続しても、
分岐回数カウンタは+1されます。
そしてこの値がリミットを超えると、セキュリティ例外となって、呼び出し
元に強制的に戻ってきます。
このリミットを設定できるのはもちろん自分の子供に対してだけであって、
自分のリミットを自分で変更することはできません。また自分の子供が
消費したカウントは、自分の消費分としてもカウントされます。

# OSECPUのセキュアの仕組み #08

(6) 無限ループ対策 [つづき]

よく無限ループ対策としてはウォッチドックタイマが使われていますが、OSECPUではこの方法を採用しませんでした。
というのは、ウォッチドックタイマに利用できそうなタイマを持っていない組み込み環境がありうると思ったからです。またウォッチドックタイマ方式だと、CPUの速さやタイマ周期によって、許されかどうかがあいまいになります。多様な環境で実行されうるOSECPUでは、これを回避したいと思いました。

なおこの無限ループ対策は、現時点でのOSECPUでは未実装です。
将来の実装を予定しています。

# OSECPUのセキュアの仕組み #09

(7) ハンドル機構

OSECPUでは、特権レベルという概念がありません。特定のコードやデータにアクセスできるかどうかは、そのポインタを知っているかどうかがすべてです。

したがってアプリは起動時にAPI呼び出し用のポインタしかもらえません。

他のポインタをもらえてしまったら、攻撃が成立してしまうからです。

しかしこのアプリから呼び出されたシステムはどうなのでしょうか。

システムはどうやって、画面などにアクセスすればいいのでしょうか。

アプリはシステムにそれらのポインタを渡せません、自分も知らないからです。だからシステムも画面にアクセスできないことになってしまいます。

この問題を解決するのがハンドル機構です。

# OSECPUのセキュアの仕組み #10

(7) ハンドル機構 [つづき]

ハンドル機構は特定のポインタを渡すことができるものの、そのポインタを使ったアクセスはすべて禁止されている特別なポインタです。
システムはアプリにハンドルも渡します。アプリはハンドルを使ってメモリアクセスすることはできません。ハンドルそのものはポインタレジスタやVPtr型のメモリに自由に読み書きできます。これでハンドルをシステムや自分の子に渡すことができます。
システムはハンドルを「アンロック」して、ポインタに戻すことができます。これで自分のワークエリアへのポインタを取得できるので、すべての秘匿情報にアクセスできます。
それならアプリだってハンドルをアンロックすればよさそうなものですが、そのハンドルはシステムによって生成されたハンドルなので、アプリにはアンロックができないのです。

# OSECPUのセキュアの仕組み #11

(7) ハンドル機構 [つづき]

ハンドルをアンロックできるのは、ハンドルを生成したプログラム自身だけです。OSECPUはJITコンパイラするたびに、翻訳ブロックIDを内部で生成するのですが、それが一致したものだけがアンロックできます。
つまり、システムのハンドルをアンロックできるのは、システム自身とシステムに静的にリンクされた関数だけなのです。
パスワードも特権レベルも何も関係ないのです。ですから特権を取りに行くような攻撃は、そもそも成立しません。
ちなみにアプリがハンドルを作れば、それはシステムからは中身が見えないということになります。そんなことができて何の役に立つのかは謎ですが（笑）。

# OSECPUのセキュアの仕組み #12

(7) ハンドル機構 [つづき]

もちろんOSECPUに内蔵のデバッガは、すべてのハンドルを自由にアンロックして中身が見えます。そうでないとデバッグがやりにくくて仕方ないです。しかしこの情報をアプリに提供することはありません。

# OSECPUの将来 #01

(1) 機能面

OSECPUは今後数年をかけて、以下を機能拡張を予定しています。

・マウスや浮動小数点演算のサポート

・ASKAの改良

・バイトコードレベルでの構造体のサポート

・タスクスケジューラ

→ これがあると、同時に複数のアプリを動かして連携させたり、
動作中のアプリの変数を読み書きできる

・メモリリーク検出支援機能

・タスクセーブ機能、タスクロード機能

→ これがあるとアプリを中断して再開できる
違う環境で再開させることも可能

# OSECPUの将来 #02

(2) 広がり

OSECPUはいろいろなところで使われることを目指しています。

Javaの標語で「Write once, run anywhere」というのがありましたが、
OSECPUはJavaよりも有望だと思います（あくまで可能性の話ですが）。
たとえばJavaは実行環境が複雑で大きいので、自作OSに載せる気にすらなりません。だから私の知る限り、どの自作OS上でもJavaアプリは動きません（Javaのミニ版のWabaなら動く自作OSはあります）。
→ この状況のどこが「run anywhere」なのでしょうか・・・

対してOSECPUは、発表して3ヶ月ほどで、自作OSに組み込まれた例が出てきています。Windows以外への移植も進んでいます。

# OSECPUの将来 #03

(2) 広がり [つづき]

OSECPUは組み込み用途でも使われるようになってほしいと思っています。
16ビットや8ビットのCPUも意識して設計してきたので、それが生かされます。
この場合JITコンパイラによる実行は無理かもしれませんが、その場合は、
クロスコンパイラで静的に実行プログラムを生成します。
アプリが共通で「使える・開発できる」のはいいことだと思います。

組み込み用ではセキュリティチェックを常にOFFにする可能性もあります。
というのは本当にメモリが少ない環境では、セキュリティ情報を保持でき
ないかもしれません。そうであれば、「全く動かないよりは、非セキュア
でも動かせるほうがマシ」となって、OFFにできるという選択肢が生きて
きます。仮にOFFにするとしても、開発時はPC上でできますので、
セキュリティチェックを受けることができ、デバッグに役立ちます。

# OSECPUの将来 #04

(3) 第三世代OSASKへ

OSECPUはそもそも、私が以前より構想を練ってきた「第三世代OSASK（おさすく）」の機能縮小版です。

i = j + 1; のような演算をするときに、この演算は本当に32ビット演算が必要なのかどうか、OSECPUのバイトコードだけでは容易に判断できません。
第三世代OSASKのバイトコードでは、すべての演算にどの規模の演算が必要なのかを明記できるので、この問題を解消できます。
同様の方法で第三世代OSASKは1～256ビットの演算に対応し、効率よいコード生成ができます。

OSECPUはおそらくある段階で第三世代OSASKへ移行します。
その際には、OSECPUアプリからコンバートするようなツールを用意します。
したがってOSECPUアプリが無駄になることはありません。

# OSECPUの将来 #05

(4) OSECPUで（最終的に）実現したいこと

世の中のソフトウェアのうちの半数以上は、限界まで高速であることを求められてはいないと思います。
それらに対しては、バグが少ないことや脆弱性がないことのほうが求められています。
そのようなソフトウェアの開発や実行の基盤になることを目指します。
ハードウェアの性能を引き出すような用途は、OSECPU以外でやればいいです。
OSECPUは地味で目立たない存在でいいし、OSECPUがすべての用途をカバーする必要はないのです。

# OSECPUの将来 #06

(4) OSECPUで（最終的に）実現したいこと [つづき]

「過ぎたるは及ばざるが如し」
Javaはいろいろとやりすぎてしまったのではないかと思います。結果として
実行環境は巨大になり、多様な環境に容易に移植できる規模では
なくなってしまいました。しかしJavaはきっと違うことを目的にしていたのだと
思うので、Javaの方針が間違っているとまでは思っていません。
OSECPUはワークステーションから組み込みマイコンまで幅広く対応し、
共通のソフトウェア資産を提供します。
新規のCPU、新規のOSが出たときにも、すぐに対応することができるでしょう。
いわば業界の水準の底上げ的な役割を果たしたいです。
最先端を追及するようなことは、それぞれのOSが目指せばいいことで
あって、OSECPUはそれらのOSの共通の底上げ資産になれれば
それでよいのです。

# OSECPUの将来 #07

(4) OSECPUで（最終的に）実現したいこと [つづき]

「実験的側面」
この単純なアーキテクチャだけでどこまでできるのかを確かめたいです。
この単純なアーキテクチャだけでもかなり機能密度が高まることを
示したいです。
しかし機能密度に関するすべては、OSECPUにとっては余興であって
本質ではありません。まあ設計がよくできていることの間接的な証では
あるかもしれませんが。

# 開発に参加しませんか？ #01

OSECPUはセキュリティキャンプの教材として開発が始まり、実際、
キャンプ中に教材として使われて、開発や改良が進んでいます。
しかしキャンプとは無関係な人も自由に開発に参加できます。
→ キャンプ生ほどの手厚いサポートは時間的に無理ですが・・・

今はまだOSECPUは足りないものだらけなので、何を作っても貢献できます。
できることをちょっとやってみるだけで十分です。

ということで、もしよかったら、Wikiのほうをのぞいてみてください。
待っています！