OSECPU-VMの資料

セキュアなシステムを作ろうクラス

OSECPU-VMゼミ

講師　川合秀実

他のゼミ、他のクラスのみなさんへ #01

OSECPU-VMゼミでは、講師が中心となってセキュアなVM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、セキュアなシステムを作ることについて具体的に学んでもらいます。

ここでは、そのOSECPU-VM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-VMとは？ #01

基本

セキュアなVM/OSを自作するというプロジェクト

「おせくぷ・ぶいえむ」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

Javaや.NETのような機種依存のないバイトコードを実行するVM

→ 独自の命令体系で、実在するCPUの命令体系ではないので、
仮想マシン（これをVMという）を作って実行させる

OSECPU-VM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-VMアプリの機能密度: 圧倒的な世界一

Windows上でOSECPU-VM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VM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

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

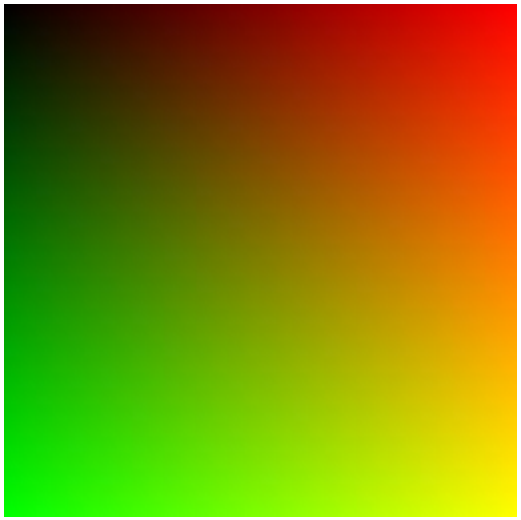
VMが何でも引き受けて、アプリに楽をさせれば、アプリが小さいのは当然
→ その代わりVMは大きくなる

両方が小さいのは本物
→ 設計がよい証拠（註：自画自賛です）

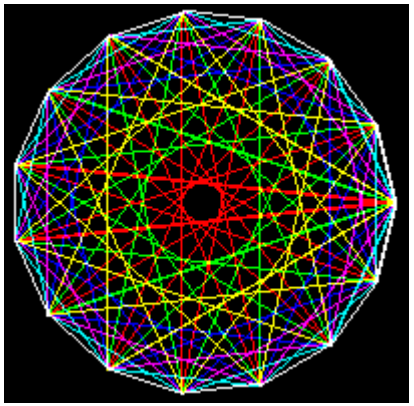
OSECPU-VMとは？ #03

ここまでのまとめ

OSECPU-VM ＝ セキュアVM技術 ＋ 機能密度技術



8バイトでグラデーション



63バイトで
こんな模様

430バイトで
こんなゲーム

2014年度の開発方針 #01

2013年度はJITコンパイラ方式でVMを作った。十分に高速に動作した。
しかし、デバッグや仕様変更は大変だった。

2014年度は、あえてVMをインタプリタ方式にする。これはJITコンパイラ方式に比べて10倍程度遅く、実用上でも問題があるレベルだが、今はこれを気にしない。2015年度以降にJITコンパイラ版を作ればいいだけのこと。

2014年度は試行錯誤を繰り返して、VMの仕様を完全に固める。
仕様が固まらないと関連ツール群の開発も進まないから。

最初からJITコンパイラ版を作ってしまった2013年度はまさに「早すぎる最適化」をやってしまったことになるが、しかし当初はJITコンパイラがどのくらい高速なのかを確かめる必要もあったので、これはこれでやむを得なかった面もある。

OSECPU-VM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-VMにおけるセキュアの考え方 #02

「セキュリティチェックはOFFにもできる」という設計

OSECPU-VMではセキュリティチェックをしないという選択を認めている。
セキュアなVMを目指しているので、これは意外に思うかもしれない。
しかしこれは矛盾ではない、むしろプラスに働いている。

セキュリティチェックを必須なもので、OFFにできないと決めてしまうと、
実行速度を高めるために、チェックを簡易なものにしたいという誘惑が
VM設計者に生じる。「やろうと思えばあれもこれも検出できるが、さすがに
そこまではやらない、だって遅くなりすぎるから」などという具合に。

OFFにすることを認めたことで、妥協は許されなくなった。「チェックがONの
ときはあえてチェックすることを選んでいるのだから、期待に応えてあれも
これもチェックしなければ」と感じるのである。

VMの優劣はどこで決まるか？ #01

VMの優劣はどこできまるか？

互換性 → Javaも.NETもOSECPU-VMも機種依存はない
実行速度 → JITコンパイラ方式にすれば、どれも同じくらい速い
アプリの書きやすさ → これはVMの優劣ではなく、関連ツールの優劣
セキュア → 重要（これがないと怖くて使えない）
機能密度 → 実は重要（でもみんなはまだ気づいていない）
VMの移植しやすさ → これも重要

互換性・実行速度・セキュアは最終的にみんな同じレベルへ向かうはずだ。
そうなると、差別化できるのは、機能密度とVMの移植のしやすさしかない。

VMの優劣はどこで決まるか？ #02

VMの移植のしやすさ

VMが移植しにくいと、自分でOSやCPUを作った場合に、その環境上で
アプリを動かすのは絶望的になる。
→ 自作OSを作った人はたくさんいるが、Javaや.NETアプリを自作OS上で
動かした例は個人レベルでは皆無。
でもOSECPU-VMアプリを動かした人はいる。

もしVMが移植できなければ、世にあふれるJavaアプリや.NETアプリは
自作OSにとっては何の役にも立たない。結局、移植しなければいけない。

→ "**Write once, run anywhere**" は、ただのスローガンだったのか？ ！

OSECPU-VMは、より広い "**Write once, run anywhere**" を目指している。

VMの優劣はどこで決まるか？ #03

機能密度

機能密度が高ければ、アプリは小さくなるので少ない容量にたくさん
詰め込める。機能密度が低いアプリではこれができない。
　→ つまりOSECPU-VMのほうが適用範囲が広い

8バイトでグラデーションが描け、63バイトであのようなボールが描ける。
この2つの画像に限って言えば、OSECPU-VMアプリは世界最高の画像
保存形式である。他にも幾何学模様は圧倒的に小さくなる。
それならば、JPEGやPNGに並ぶもう一つの画像形式としてアピールできない
だろうか・・・できる！

もちろんその際はAPIから描画関係以外をすべて削る。これでOK。
もともとセキュアに作ってあるので、何の心配もない。機種依存もないので
どのOS向けにも容易にビューアが作れる。

VMの優劣はどこで決まるか？ #04

機能密度（つづき）

同じアピールは、もちろんJavaでも.NETでもできるだろう。しかし画像形式の優劣は、画質とサイズである。画質が同じならサイズしかない。
Javaアプリや.NETアプリがOSECPU-VMアプリよりも大きい以上、もはや彼らに勝ち目はないのだ。

おそらくこの傾向は画像形式にとどまらない。他にもOSECPU-VMアプリの方が専用形式に勝ってしまうことがありうる。そうなれば、そこにも進出のチャンスがある。
こういうことは、OSECPU-VMアプリの圧倒的な機能密度のおかげであって機能密度を軽視してきた世間一般のプログラマは気づいていない。

OSASK計画における機能密度の歴史 #01

機能密度改善の実績

OSECPU-VMはOSASK計画の最新作品でもある。

普通、VMやOSがバージョンアップすると、機能は増えるが、それにつれてアプリのサイズも増える傾向がある。しかしOSASK計画はそうではなく、機能密度も改善されていくので、バージョンアップにつれて小さくなってきた。

	はりぼてOS	OSASK-G1	OSECPU-r1	OSECPU-r2	比較対象
グラデーション	不明	不明	13	8	270, Java
bball	350	186	71	63	114, MS-DOS
インベーダー	1509	1108	507	430	2192, Win9X

単位はすべてバイト。　OSASK-G1 と OSECPU-r1 の間には8年のブランク。
比較対象は、OSASK計画以外での世界最小記録のもの。

アプリのサイズとはなんなのか？ #01

アプリのサイズについて考えていること

アプリは、そのアプリをそのOS上で実現するために必要な情報をまとめたものだと考えることができる。なぜこれほどの差が出るのか。

たとえばbballを例にしよう。bballを描画するために、どうしても不可欠な情報が存在するだろう。そのほかにOSやVMに依存した情報もあるだろう。この環境依存な部分をどれだけ小さくできるかが、結果を左右しているのだと思われる。

bballを見ていると、不可欠な情報は63バイトかもしくはそれ未満である。となれば「はりぼてOS」版のbballアプリの287バイト以上は、ムダみたいなものである。少なくとも本質ではない（ムダは言いすぎだが）。本質以外に 287÷63＝4.55 なので、実に4倍も本質以外の情報を持たされている。

アプリのサイズとはなんなのか？ #02

アプリのサイズについて考えていること（つづき）

「はりぼてOS」はOS自作入門で作る教材用のOSなので、それくらいの無駄はあるかもしれない。しかし世界第二位のMS-DOS版ですら、114バイトのうちの44%以上は本質以外の無駄である。

世間のプログラムのほとんどは、世界一位でも二位でもない。というか「はりぼてOS」の350バイトという結果でさえ、WindowsやLinuxから見ればとても立派なものである。

世間のプログラムの本質は、一体どれほどなのだろうか・・・。

OSECPU-ASKA入門 #01

(0) はじめに

以下の記事はosecpu122d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。
→ ちなみに以下のページを読めばrev2の最新版が見つかります。
http://osecpu.osask.jp/wiki/?page0074

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(MODE_COL3, 7, 640, 480, 0, 0);
    junkApi_fillOval(MODE_COL3, 1, 300, 300, 320-150, 240-150);
```

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らないでしょう。
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-VM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app0101.askとする）、Windowsのコマンドプロンプトで、

```
prompt>amake app0101
```

OSECPU-ASKA入門 #03

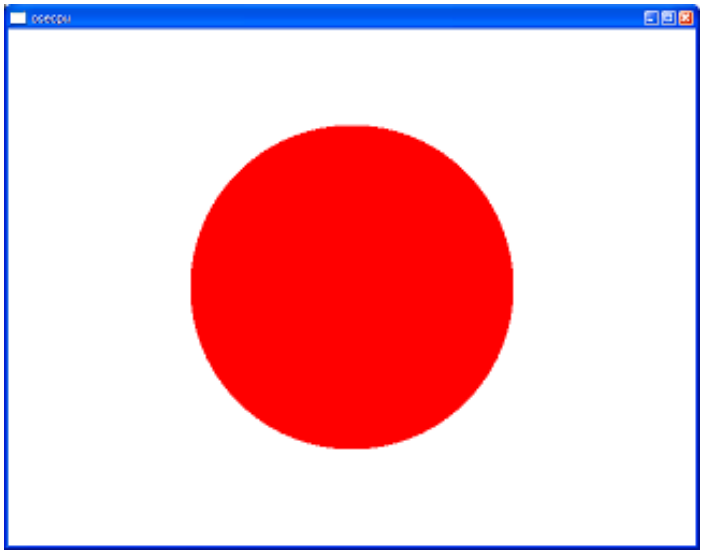
(1) ASKAは意外に難しくない？ [つづき]

と入力すると、app0101_.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 app0101_.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。

そしてapp0101_.oseのサイズを確認します。・・・なんと14バイトです。これがOSECPU-VMの実力です。

（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/?page0087 ）

スタックマシン vs レジスタマシン #01

スタックマシン:

たとえば a=((v+x)+(y+z))/4; を計算するときに、

```
push(v); push(x); add();
push(y); push(z); add(); add();
push(4); div();
pop(a);
```

という機械語を使っているのがスタックマシン。スタックに積んで演算する。

addやdivなどの演算命令に引数がないところが特徴。

レジスタマシン:

```
load(r0,v); load(r1,x); add(r0,r1);
load(r1,y); load(r2,z); add(r1,r2); add(r0,r1);
load(r1,4); div(r0,r1); store(a,r0);
```

こんな感じなのがレジスタマシン。レジスタに読み込んで演算する。

スタックマシン vs レジスタマシン #02

世間で言われていること:

スタックマシンは、アプリのバイトコードが小さくなる。

レジスタマシンは、動作が高速。

見た目はそうかもしれないけど・・・・本当にそうだろうか？

RISC-CPUのレジスタマシン:

add(r4,r0,r1); add(r5,r2,r3); add(r4,r4,r5); load(r5,4); div(r4,r4,r5);

RISCはレジスタがたくさんあるので、変数をレジスタに割り当てる。

v→r0, x→r1, y→r2, z→r3, a→r4

これならloadやstoreがかなり減らせる。

これなら5命令しかないので、スタックマシンより小さいのでは？

スタックマシン vs レジスタマシン #03

簡単な実験:

push, pop, add, div をすべて1バイトの命令コードとする。
レジスタ番号や変数番号も定数4もすべて1バイトとする。

スタックマシン → 16バイト
RISC風レジスタマシン → 19バイト

・・・本当だ、RISC風でも負けている・・・。

しかし:

実際のプログラムでは、z=x+y; 程度の簡単な演算が多い。
push(x); push(y); add(); pop(z); → 7バイト
add(r2,r0,r1); → 4バイト

・・・おお！？

スタックマシン vs レジスタマシン #04

ではOSECPU-VMはどうするか:

もっと厳密な実験をしてみても、スタックマシンの優位性は「式が複雑な場合のみ」だと判明。そしてそんな式はたまにしか出てこない。

ということで、OSECPU-VMはレジスタマシンを採用。

レジスタ数は整数レジスタだけでも64本！（load/storeを減らしたい）

命令セット的には、もっとレジスタを増やす余裕がある。

まとめ:

JavaVM : スタックマシン

.NET-VM : スタックマシン

OSECPU-VM : レジスタマシン

ということで、OSECPU-VMはVM界の異端児。

OSECPU-VMの内部仕様 #01

(1) 整数レジスタ

OSECPU-VM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、JITコンパイル版では実際の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-VM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～R27（40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。

R28～R2B（ 4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。

R2C～R2F（ 4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R28～R2Bはアプリが自由に用途を決定できます。

OSECPU-VM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-VMのレジスタはかなり多いほうだと思います。これはOSECPU-VM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-VM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-VM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-VMの内部仕様 #05

(3) ポインタレジスタ

OSECPU-VM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～P27, P28～P2B, P2C～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も持っていて、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。

OSECPU-VMの内部仕様 #06

(3) ポインタレジスタ（つづき）

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。

初期化し忘れていたのに、メモリを読んだ場合もセキュリティ例外です。

Int8sなメモリに200を書き込もうとしたら例外ですし、Int16uなメモリに負の数を書き込もうとしても例外です。

これらの仕組みによって、多くのやっかいなバグや脆弱性を未然に防げます。

OSECPU-VMの内部仕様 #07

(4) セキュリティ例外

OSECPU-VM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ができます。

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-VMの内部仕様 #08

(5) JITCというAPI

OSECPU-VM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。

つまり後付けで動的に関数を作るAPIだと考えたらわかりやすいかもしれません。

これにより、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。

この作った関数とアプリとは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-VMの内部仕様 #09

(5) JIT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-VMには大きな可能性があります。

たとえばOSECPU-VM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-VMならOSECPU-VMのバイトコードさえ生成できれば、あとはOSECPU-VMが面倒を見てくれます。しかもOSECPU-VM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-VMの内部仕様 #10

(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-VMの内部仕様 #11

(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-VMの内部仕様 #12

(5) JITCというAPI [つづき]

OSECPU-VM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-VMのバイトコードを作ればいいのです。

現在、OSECPU-VM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-VMのセキュアの仕組み #01

(1) 整数レジスタは厳密にはチェックしない

OSECPU-VM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。
・それぞれのレジスタについて上限と下限を設定するのはかなり面倒。

ただし演算時には想定ビット数を示すことができ、そのビット数に収まらない結果になった時には、セキュリティ例外で停止してくれます。
オーバーフローやアンダーフローを早期に発見できます。

OSECPU-VM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-VMでは「死んでいる」という

OSECPU-VM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ビット以上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-VM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-VM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-VM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P2F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-VMのセキュアの仕組み #06

(5) 不正な分岐への対策 [つづき]

OSECPU-VM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-VMの内部仕様 #11」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P2F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P2F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-VMのセキュアの仕組み #07

(6) 無限ループ対策

OSECPU-VM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-VMでは、実行するたびに、命令実行数カウンタという内部変数が
+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-VM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タイマ割り込みが使われていますが、
OSECPU-VMではこの方法を採用しませんでした。
というのは、タイマ割り込み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。またタイマ割り込み方式
だと、CPUの速さやタイマ周期によって、許される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-VMでは、そのような
あいまいさを回避したいと思いました。

OSECPU-VMのセキュアの仕組み #09

(7) ハンドル機構

OSECPU-VM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-VM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-VM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-VMはJITコンパイルするたびに、翻訳ブロックIDを内部で生成するのですが、それが一致する関数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-VM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-VM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-VMのセキュアの仕組み #13

(8) ファイルアクセスやネットワーク関係

OSECPU-VMアプリは、自由なファイルアクセスを許されていません。
アクセスできるのはインストールディレクトリなどの限定的な範囲だけです。
しかしこのような制限があると、他のフォルダのデータを読ませたいときに
非常に不便です。

これに対処するため、コマンドラインで明示されているファイルに限り、
許可されていないフォルダ内であってもアクセスを許可します。しかし
その際も、ファイル名はアプリには明かされません。第2引数のファイルを
開く、のように指定します。コマンドライン引数も自由には取得できず、
ファイル名かもしれないものは隠されます。

以上はCUIの場合ですが、GUIの場合は、ファイル選択ダイアログで選択
したファイルだけが例外的に許されるようになります。

OSECPU-VMのセキュアの仕組み #14

(8) ファイルアクセスやネットワーク関係 [つづき]
ネットワークについても似たような仕組みを予定しています。
つまりサーバ名やURLを入力しなければいけないのです。

OSECPU-VMでは、
「アプリがどこそこへの接続を要求しています、許可しますか？」とか
「アプリがどこそこのファイルへのリードアクセスを要求しています、
許可しますか？」
みたいな確認を促すことはしません。こんなのはOKを押す習慣ができる
だけです。そうではなくて、明確な入力をさせるのです。

毎回入力を促されたら、面倒な場合もあるでしょう。ですから二度目からは
問い合わせしないで前回入力値を使うというオプションは検討します。
しかしこのオプションをデフォルトでONすることはアプリにはできません。

OSECPU-VMのセキュアの仕組み #15

(8) ファイルアクセスやネットワーク関係 [つづき]

ネットワークゲームなどでは、特定のサーバに固定的につながせたいかもしれません。それでもユーザが入力する必要があります。

アプリは「推奨されるサーバは〇〇です」と入力ダイアログに表示させることはできますし、それをユーザがコピー＆ペーストで入力することも許しますが、しかし入力初期値にすることは許しません。つまり、「Enterを押すだけ」という状態にはしないのです。

ファイル選択のダイアログについても、推奨されるファイルパスを表示することはできても、アプリから入力初期値を設定可能にはしません。

いずれにせよ、面倒なのは最初の一回だけです。それ以降は、問い合わせをユーザの意志で回避できます。ですからこれくらいの不便さは許容されると思っています。

OSECPU-VMのセキュアの仕組み #16

(8) ファイルアクセスやネットワーク関係 [つづき]

このような方法は、他のOSやVMなら「権限」で解決するところです。
つまり、そのサーバへのアクセス権がありません、みたいな方法で、
サーバやファイルを守ろうとするのです。しかしこれは権限さえあれば、
ユーザの知らないうちに悪事を働くことが可能だということです。

OSECPU-VMの方法は、権限なんて関係ありません。とにかくユーザの
知らないところに勝手にアクセスする、ということを許さないのです。

アプリは、推奨する接続先を示すことができますが、しかし、その接続先を
選んでくれたのかどうかは、結局わかりません。ユーザはダミーのサーバ
を指定したかもしれません。ですから、接続先が特定の場所なら、その時
だけ悪事を行う、ということも容易ではないでしょう。

OSECPU-VMの将来 #01

(1) 機能面

OSECPU-VMは今後数年をかけて、以下の機能拡張を予定しています。

・マウスのサポート

・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-VMの将来 #02

(2) OSECPU-VM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-VM以外でやればいいです。
OSECPU-VMは地味で目立たない存在でいいし、OSECPU-VMがすべての用途をカバーする必要はないのです。

OSECPU-VMの将来 #03

(2) OSECPU-VM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-VM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-VM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-VMの将来 #04

(3) 夢物語

OSECPU-VMは、とにかくアプリのサイズが世界一に小さいので、いつかきっと注目されます。残念ながらセキュアでは世界一とかそういうことはないと思うので、そちらで注目を集めることはないと思います。

その時には、私はこれが「セキュリティキャンプの成果でもある」と主張します。そして開発に関わってくれた人たちもきっと評価されるでしょう。

OSECPU-VMは世界を変える「可能性」を持っていると私は思います。
まあ分からない人にはわからないかもしれないです。笑いたい奴には
笑わせておけばいいのです。どうせ最後に勝つのは私たちなのですから。

OSECPU-VMの将来 #05

(3) 夢物語 [つづき]

OSECPU-VMでアプリを書いて遊んでいると、本当になんでも小さくなってしまうことに驚かされます。今まで自分が作ってきたプログラムは何だったのだと思います。私の作ったものの半分以上は無駄で構成されていたのです。私はそう感じます。

感動します。世界観が変わります。この体験をしたことがあるかどうかで、普通の人たちよりも一歩先の世界が見えるかもしれません。

そしてできるだけ無駄のないプログラムを書きたいと感じます。そのためには現状ではOSECPU-VMしか選択肢がありません。それくらい圧倒的な差なのです。・・・もちろん未来にはもっと選択肢が増えてほしいです。

OSECPU-VMの将来 #06

(3) 夢物語 [つづき]

OSECPU-VMを理解するために、微分や積分は必要ありません。というか
小学校の算数以上のことはやっていない気がします。英語が読めなくても
問題ありません。そこそこのプログラムが書ければ十分です。OS自作入門
が理解できたのなら、もはや余裕だと思います。

そんな簡単なことだけで圧倒的な世界一を達成して、さらに世界を変える
可能性まであるのです。これよりもいい話があるでしょうか。
このテキストを読んだのもきっとチャンスです。それを無駄にしていいのでしょ
うか。

数年後には開発者も増えて、もう主要な開発者に簡単にはなれないかもしれ
ません・・・。

開発に参加しませんか？ #01

OSECPU-VM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-VM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！