

いまどきのデバッグテクニック

~ たこ焼き の作り方 ~
(の話を中心に)

解析魔法少女 美咲ちゃん マジカルオープン!

やばう級部ゲーム4巻



クラッカーズ

倉塚美咲が「キミだけ」に教える
クラックの常套テクニック

アセンブラの入門書としてもオススメ
（1500円）

西村システム

あるソフトを契機として買収の話が
持ち上がっていた。

プログラマは敵前逃亡



デバッグが困難

任務を遂行するためには
その(誰かが作った海のものとも
山のものともわからない)プログラ
ムを修正してリリースする必要が
あった

そのソフトとは

ズバリ！

K馬ソフト

これを可及的すみやかに
bug fixすることが与えられた使命

デバッグの基本的な方針

- バグに対して仮説を立てる
- それを検証する

それは、たこ焼きで言えば、
焼けたかどうかを確認するために
1秒ごとにクシを刺していても
たこ焼きが穴だらけになってしまう
ことに似ている。

テクニック1. APIのhooking

- ・Windows上で動くプログラムは、ほぼ間違いなくWindowsAPIを呼び出すことで機能を実現している。
- ・特定APIの呼び出しをinterceptできれば、特定APIを呼び出したときのパラメータを記録して、そのログを書き出したりできる！

APIのhookingとは

たこ焼きで言えば、たこ焼き生地をたこ焼きプレートに載せる動作を横取りし、その瞬間に「てんかす」を入れてしまう技術だ。

偽のDLLの作成

実行ファイル

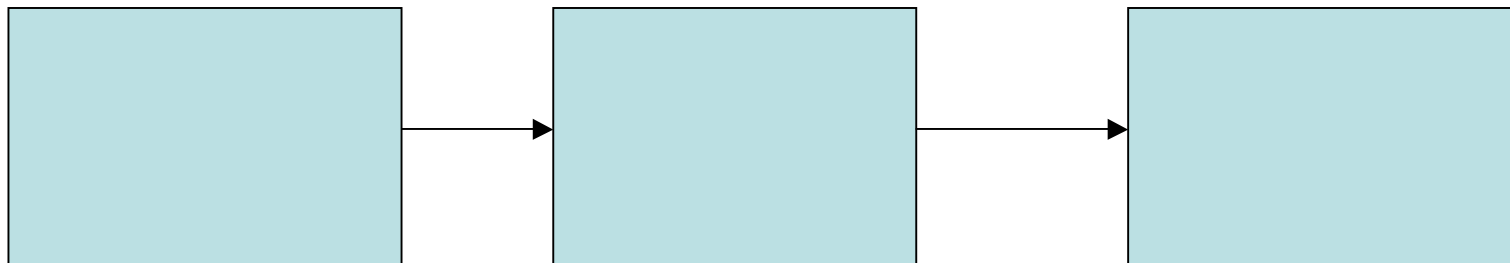
本来のDLL



実行ファイル

偽のDLL

本来のDLL



偽のDLLをどうやって作るか？

DLLからexport tableを列挙するツールを作る(使う)

DLL export tableの列挙ツール

<http://www.chiyocloner.net/>

のlistexp.exeとlistexp2.exe

```
__declspec(naked) d_GetLocalTime()  
{ _asm { jmp p_GetLocalTime }}
```

この行を↑をコメントアウトして..

```
VOID WINAPI d_GetLocalTime(LPSYSTEMTIME pTime) {  
  
}
```

偽のDLLをどこに置くか？

- ・絶対パスで指定した場合は
その DLL がロードされる。
Side-by-Side の介入の余地はない。
- ・パスを指定されなかったとき
 - 1.アプリケーションの exe ファイルが存在するディレクトリ
 - 2.システムディレクトリ
 - 3.16-bit システムディレクトリ
 - 4.Windows ディレクトリ
 - 5.カレントディレクトリ(WindowsXP SP1/Windows2003 SP3以降。以前は
カレントディレクトリは2.番目であった)

参考URL)

<http://d.hatena.ne.jp/NyaRuRu/20040614#p3>

API hijackの仕組み

- 実行ファイルEXE / DLLのフォーマットはPE(Portable Executable)フォーマットである。
- Win32プロセスは、このPEフォーマットのファイルをほぼそのままメモリに読み込んで実行している。

(参考書籍 : 『Linkers & Loaders』
ISBN: 4274064379)

PE loaderの仕組み

- PE フォーマットのバイナリでは API コールはすべて間接ジャンプになっている。
- .rdata というセクションにそのジャンプ先の値を書き込んだテーブルが収められている。
- コンパイル後のアセンブラでは `call .rdata[10]` って感じになっていて、.rdata 領域に API の実アドレスが並んでいる。
- Windows の PE ロードは .rdata セクションのアドレステーブル中の値を書き換えることで、ロードされたバイナリと API の実コードのある空間とをリンクする。

参考URL)

<http://d.hatena.ne.jp/nitoyon/20040109>

.rdata (import section)
.text
.data

API hijack用ツール

API hijack

- <http://www.codeproject.com/dll/apihijack.asp>

を使うと良い

```

SDLLHook D3DHook =
{
    "DDRAW.DLL",
    false, NULL,    // Default hook disabled, NULL function pointer.
    {
        { "DirectDrawCreate", MyDirectDrawCreate },
        { NULL, NULL }
    }
};

BOOL APIENTRY DllMain( HINSTANCE hModule, DWORD fdwReason, LPVOID lpReserved)
{
    if ( fdwReason == DLL_PROCESS_ATTACH ) // When initializing....
    {
        hDLL = hModule;

        DisableThreadLibraryCalls( hModule );

        GetModuleFileName( GetModuleHandle( NULL ), Work, sizeof(Work) );
        PathStripPath( Work );

        if ( strcmp( Work, "myhooktarget.exe" ) == 0 )
            HookAPICalls( &D3DHook );
    }

    return TRUE;
}

```

さきほどのエロ本



テクニック2. 逆実行

現代のデバッガに望まれる機能(一例)

- プログラムの逆実行(バックトレース)がしたい
- 時系列call stackの表示がしたい

→ 現在の普通のデバッガでは不可能

逆実行とは

たこ焼きで言えば、客が来ないので中までカリカリになるほど焼けすぎてしまったたこ焼きを、口どけまるやかな、ふっくらな状態に戻すことだ。

逆実行は可能か？

- 巻き戻しのために一命令ずつログを残していくと膨大な量になるし実行時間も現実的な時間で収まらない。
- 関数ごとにログを残していけば、関数単位で巻き戻すようなことは出来そう。→ 未来の言語や未来のデバッガでは可能になると思われる

逆実行の代わりに トレースログの作成をする

- OllyDbg(フリーのデバッガ)を用いてrun trace(実行トレース)を行えば良い。

欠点)

- run traceは一命令ずつの実行になるのでログに書き出したい部分がある程度絞って行なわないと莫大な量のログが生成される
- OllyDbgでは、SYSENTER命令から先をrun traceすることが出来ない。

SYSENTERとは何ぞや？

【カーネルモード(ring0)へ移行する方法】

- WindowsNT/2000 int 2E(割込みゲート)
- WindowsXP/2003(x86版) sysenter
- WindowsXP/2003(x64版) syscall
- Windows95/98/Me call(コールゲート)

int 2E(割込みゲート)とcall(コールゲート)は32ビット世代の最初のCPUである386から利用可能な方法です。

これに対し、sysenterはインテルがPentiumIIで導入した命令、syscallはAMDがK6で導入した命令です。なおsyscallはEM64Tでも利用可能です。

参考URL)

http://homepage3.nifty.com/marbacka/asm64/antekning/int2e_sysenter_syscall.html

SYSENTERの問題点

- ・呼び出し元をスタック上に記録していないので(デバッガがこの命令に対応していないと)呼び出し元がわからない。

GDT(グローバル・ディスクリプタテーブル)に入っている。

典型的なメッセージループすら run trace出来ない

```
while (GetMessage(&msg, NULL, 0, 0)>0) {  
    TranslateMessage(&msg);  
    DispatchMessage(&msg);  
    // ring0からコールバックされる  
}
```

SYSENTERを用いないように
Windows Kernelをcrackしてしまう？

→ 出来なくはないのだが
Windows自体の
リバースエンジニアリングは禁じられてい
るので..

→ VirtualPC のなかにWindows2000をイ
ンストールしてそこでログをとることに。

さきほどのエロ本



そして..

API hooking と run traceとを組み合わせ
わせて、特定APIの呼び出しログをと
ることに成功した！

テクニック3. 呼び出し元の特定テクニック

いまAとB,Cの3つの関数から
呼び出される関数Fの中で
関数Aから呼び出されたときのみ
データをログに書き出したい。

呼び出し元関数の特定とは

たこ焼きで言えば、

F = たこ焼き屋の親父

A = 10,000円札を出す客

B = 10,00円札を出す客

C = 小銭を出す客

たこ焼き屋の親父としては、10,000円札を出す客には要注意だ！たこ焼きを買うのに10,000円札を出すだなんて、なんか怪しい。これは偽札かも知れないぞ！

関数Fを呼び出されたとき

```
void __cdecl F(int a,int b){  
}
```

関数fがstdcallかcdeclならば
スタックの状態はこうなっているはず！

ESP →	戻り先アドレス
	引数a
	引数b

関数Aが関数Fを呼び出すのなら
関数Aを書き換えてそこでログを書き
出すようにはしないのは何故か？

関数Aに対応するソースコードが手元にあると
は限らない。手元にある関数Aはすでにバイ
ナリコードになっているかも知れない。そして、
それは暗号化されているかも知れない。

そう。(いまは)関数Aは不可侵の領域にある
のだ。

呼び出し元を知るためには呼び出し規約を理解している必要がある

- ・ cdecl

WindowsAPIではこれが一般的。

呼び出し元でスタックを調整する方式。

- ・ stdcall

呼び出された関数側でスタックを調整する方式。

- ・ fastcall

引数をスタックに積まず、レジスタ渡しする方式

その(スタック上の)戻り先アドレスが 関数Aであるかを調べるには？

これには可搬性のある方法が無い。
関数がソースコードに記述した順番
で実行コードに落ちると仮定して
良いなら 右図のようにしておい
て

```
void funcA(){  
    // 処理  
}  
void funcB(){ }
```

戻り先アドレスが&funcA と&funcB
の間の数値かどうかを調べれば
良い。

DLL上の関数ならば、
GetProcAddressというAPIでエ
クスポート済み関数アドレスが取
得できる。

関数Aから呼び出されたときだけログを書き出すことが出来るようになった！！

ポイント

これをさきほどのAPIのhookingと組み合わせれば、特定の関数から特定のAPIが呼び出されたときにのみログを書き出すことが出来る。

テクニック4. マイクロスレッドによる バグの再現

再現性のないバグをどうやって確実に再現させるか？

たとえば、マルチスレッドのプログラムで排他処理に何らかのバグがあると、1万回に1回だとか、10万回に1回だとか極めて低い確率で(だけど長時間動かしていれば確実に)バグが露呈することがある。

マルチスレッドのバグが なぜやっかいなのか？

たこ焼きで言えば、マルチスレッド
で走るプログラムだと、たこ焼きを
焼こうとしたときに、そのたこ焼き
プレートが誰かが横取りして勝手に
焼きおにぎりを焼くのに使って
しまうのだ。

焼きおにぎりを焼くのに使ったたこ焼きプレートは焼きおにぎりを焼くのが終われば返してくれるのだが、キッチンと洗わずに返しやがる時がある。
(これがバグにつながる)

想像図



焼きおにぎり焼き

再現性がないとデバッグが困難

再現性のあるバグの場合、以下のようにすれば原因を特定できる。

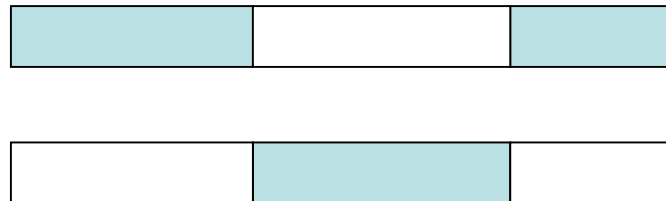
```
void draw( ){  
    static int n = 0;  
    if (++n == 100) {  
        *(int*)(0) = 0; // アクセス違反を起こすコード  
    }  
    // 実描画処理  
}
```

マルチスレッドで動くプログラムを 並列型から並行型へ書き換える

- 並列(parallel)型



- 並行(concurrent)型



マイクロスレッドとは？

- 並行スレッドの一種

```
void somefunc( ) {  
    while(true) {  
        a( );  
        yield(); // これで次のマイクロスレッドに切り替わる。  
        b();  
        yield(); // これで次のマイクロスレッドに切り替わる。  
    }  
}
```

マイクロスレッド？

どこかで見たことある概念だなあ？

- Lispの継続(continuation)
- Ruby/C#のyieldで実現されるコルーチン(coroutine)
- Windowsのfiber(「非時分割で非プリエンプティブ(協調型)なスレッド」)

マイクロスレッドは どうやって実装するのか？

関数の呼び出し履歴や、ローカル変数はスタック上にある。

実装案1)

yield呼び出し時にスタック領域をコピー

→ 非常に時間がかかる

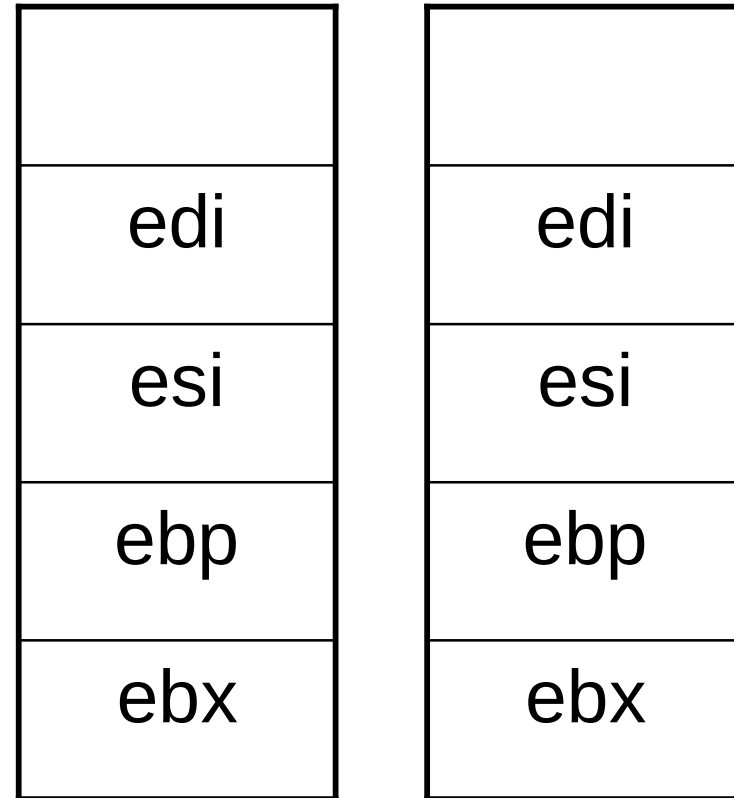
実装案2)

スタックポインタのみ切り替え

→ これを採用する！

yield(マイクロスレッドの スイッチング)の実装

```
asm {  
    push ebx  
    push ebp  
    push esi  
    push edi  
    mov eax,[this]  
    xchg esp,[eax].register_esp  
    pop  edi  
    pop  esi  
    pop  ebp  
    pop  ebx  
}
```



スタックフレーム切り替え型の マイクロスレッド実装上の注意点

例外が正しく処理されない

→ Visual C++は例外処理の実装にWin32 SEH(Structured Exception Handling)を用いている。

そこで、スタックフレームを単純に切り替えただけではダメで、SEHが正常に動作する条件を満たさなければならない。

SEHが正しく動作する条件とは？

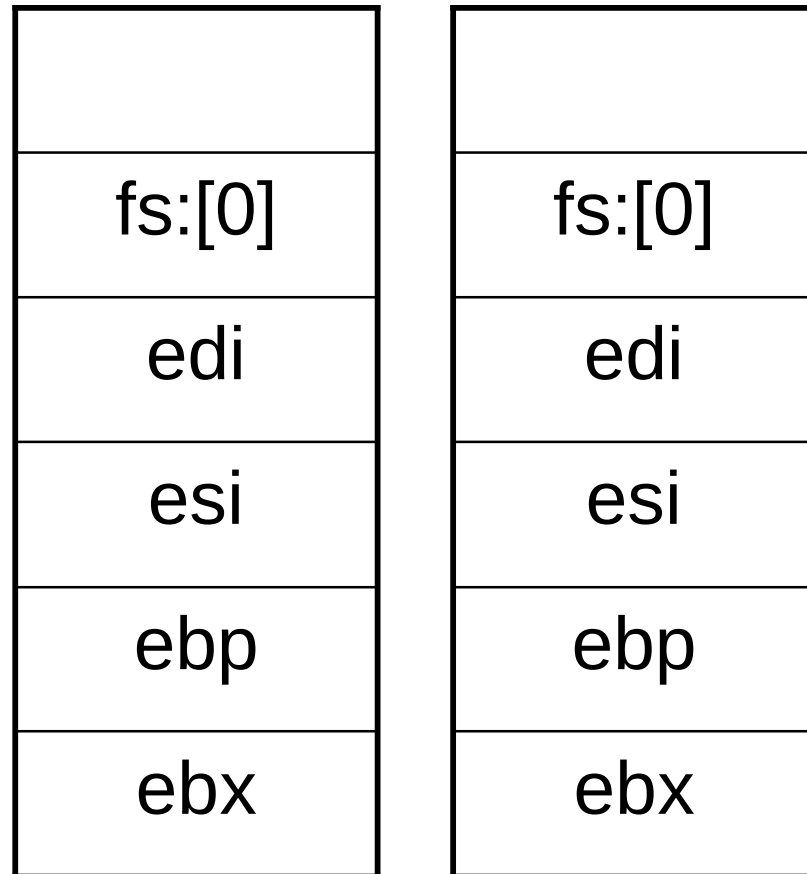
fs:[0] が例外が起きたときに呼び出すべき関数
ポインタ(正確には、呼び出すべき関数のリン
クリスト)

レジスタfsはTIB(Thread Information
Block)を指している

よってfs:[0]も保存しなければならない。

fs:[0]もスレッドをスイッチするときに
保存することによりSEHの
動作条件を満たす。

```
asm {  
    push ebx  
    push ebp  
    push esi  
    push edi  
    push dword ptr fs:[0] // 例外処理の  
    ために必要  
    mov eax,[this]  
    xchg esp,[eax].register_esp  
    pop  dword ptr fs:[0] // 例外処理の  
    ために必要  
    pop  edi  
    pop  esi  
    pop  ebp  
    pop  ebx  
}
```



SEHが正しく動作するための もう一つの大切な条件

例外が発生したときのスタックフレームがOSが管理する(スタック)領域でなければならない。

理由)

「現在のOSスレッドが知っているスタック範囲外のような怪しいポインタを見つけると、例外のリンクリストを移動するOSハンドラが例外の追跡をあきらめてしまう」

参考書籍)

- 参考書籍：『GameProgrammingGems2』ISBN: 4939007332
- 参考書籍：『Windowsプロフェッショナルゲームプログラミング2』ISBN: 4798006033

マイクロスレッドを用いて プログラムを書き換え

```
void workerThread() {  
    while (true){  
        yield();  
        a();  
        yield();  
        b();  
        yield();  
        c();  
    }  
}
```

```
void runningThread() {  
    while (true){  
        MicroThread* p =  
            workerThreadをひとつ  
            ランダム(決定性アル  
            ゴリズムで)に選ぶ();  
        p->resume();  
    }  
}
```

バグに再現性(決定性)があれば あとは追いかけてやすい

```
void runningThread() {  
    while (true){  
        static int n = 0;  
        if (++n == 10000) {  
            *(int*)(0) = 0; // ここにブレークポイントを仕掛ける  
        }  
        MicroThread* p = workerThreadをひとつランダムに選ぶ  
        ();  
        p->resume();  
    }  
}
```

今回の手法まとめ

- ・APIのhooking
- ・逆実行
- ・呼び出し元関数の判定
- ・バグに再現性を持たせるための並行スレッド化

この4つの技法により、デバッグは成功した！
プログラムは修正されたのだ！！

そして究極の
K馬ソフトが完成したのだ！

おまけ

デバッグを有利に進めるコツ

- なるべくたくさんのhack(手法)を知っていること (hackには当たり外れがあるので)