いまどきのデバッグテクニック

～インスタント焼きそばの作り方～
(の話を中心に)

任務を遂行するためには その(海のものとも山のものともわからない誰かが作った)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進め方

- バグに対して仮説を立てる
- それを検証する

テクニック1.
APIのhooking

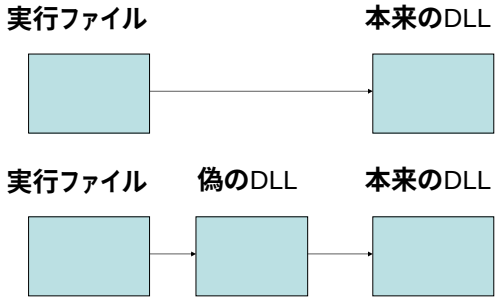
・Windows上で動くプログラムは、ほぼ間違いなくWindowsAPIを呼び出すことで機能を実現している。

・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

ひとことで言えば、焼きソバにお湯をかけるときに、お湯をかける動作を横取りし(横から眺め)、きちんとお湯が注がれているか調べることを可能にする技術だ。

偽のDLLの作成



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.chiyoclone.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

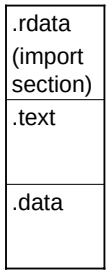
- 実行ファイルEXE / DLLのフォーマットはPE(Portable Executable)フォーマットである。
- Win32プロセスは、このPEフォーマットのファイルをほぼそのままメモリに読み込んで実行している。

(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 ローダは .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



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

    return TRUE;
}
```

テクニック2.
SYSENTERの周辺

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

ひとことで言えば、時間が経ちすぎてのびた焼きソバを何分か前の状態に戻すことだ。

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には？

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

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を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/anteckning/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。

そして..

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

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ている関数Fのなかで変数の値をログに書き出したい。

ただしログを書き出すのは、関数Aから呼び出されているときだけにしたい。

呼び出し元関数の特定とは

ひとことで言ってしまえば、

F = お湯を注ぐ関数

A = インスタント焼きソバを作る関数

B = カップスープを作る関数

C = インスタントラーメンを作る関数

インスタント焼きそばを作る関数Aから、お湯を注ぐ関数Fが呼び出されたときにだけ何かをするための技術だ。

何故関数Aのなかから関数Fを呼び出すときにログを書き出すようにはしないのか？

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

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

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

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。

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

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DLL injectionやAPI hijackと組み合わせれば、任意の関数から該当する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バグがあると、1万回に1回だとか、10万回に一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？

ひとことで言えば、マルチスレッドで走るプログラムだと、インスタント焼きソバのためにお湯を注いでいるときに、そのお湯を横取りして誰かが勝手にカップスープに注いでしまう。(その結果、インスタント焼きそばのお湯が足りなくなってしまうかも知れない！)

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。

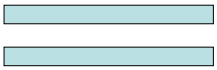
```
void draw( ){
   static int n = 0;
   if (++n == 100) {
       *(int*)(0) = 0; // アクセス違反を起こすコード
   }
   // 実描画処理
}
```

バグに再現性(決定性)があればあとは追いかけやすい

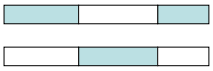
```
void runningThread() {
  while (true){
     static int n = 0;
     if (++n == 10000) {
         *(int*)(0) = 0; // ここにブレークポイントを仕掛ける
     }
     MicroThread* p = workerThreadをひとつランダムに(決定性アルゴリズムで)選ぶ();
     p->resume();    // 該当スレッドを一定時間動かすメソッド
  }
}
```

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ぁ？

- 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」)

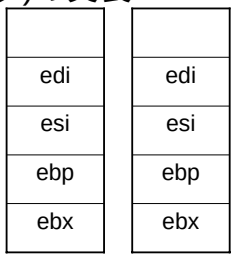
マイクロスレッドは
どうやって実装するのか？

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。

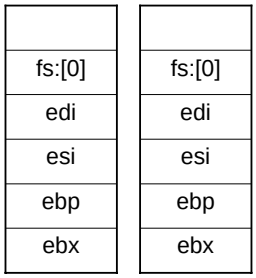
SEHが正しく動作する条件とは？

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

- dll injection / API hijackによるhooking
- sysenterの周辺
- 呼び出し元関数の判定
- バグに再現性を持たせるためのmicro thread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

- できるだけ多くの手段を知っていること
- 少なくとも、ひとつの目的に対して、一つ目の方法がダメだったときの代替手段としてもう一つの方法を持っていること

そうです。
昔の人は言いました..

転ばぬ先の

Two Way

おあとがよろしいようで..