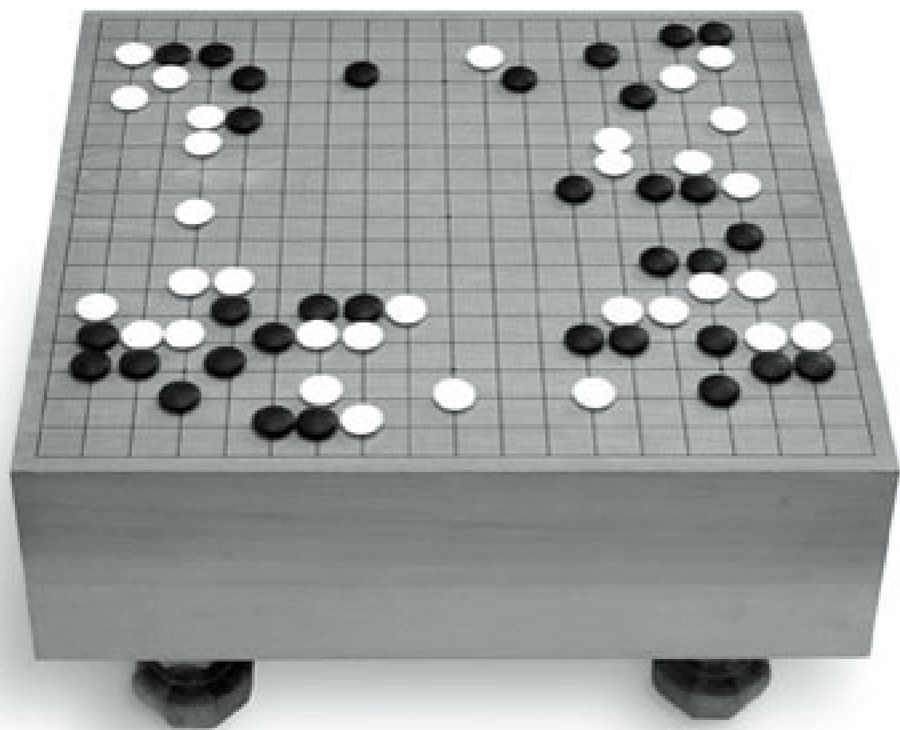
BINARY HACKS™

ハッカー秘伝のテクニック100選



高林 哲、鵜飼 文敏
佐藤 祐介、浜地 慎一郎　著
首藤 一幸

O'REILLY®
オライリー・ジャパン

本書に寄せて

2005年も暮れにかかる頃、本書の執筆者の高林氏らが中心となって「Binary 2.0カンファレンス」という会議を開催するという話を聞き、その洒落っ気に思わずニヤリとした。表面的には、流行のWeb 2.0や軽量言語のフォーマットで低レベル技術を語る、というミスマッチを楽しむジョーク企画に見えるが、その位置付けにはなかなかに深い意味がある。

コンピュータの高性能化と共に、プログラミングを取り巻く環境も大きく変化した。プログラムを書くためにまずエディタとコンパイラを作った、などというのは昔話としても色あせてしまった。現代のプログラマは、最初から強力な道具をふんだんに使える。アイデアさえあれば、軽量スクリプト言語、出来合いのライブラリ、ネットワーク上の各種サービスを組み合わせて、気の利いたサービスを素早くローンチすることができる。

最先端の道具を使いこなして優れたアイディアを素早く実装してゆくのは、いかにもスマートだ。一方で、この時代に、コンパイラの吐き出すバイナリを調べたり、実行中のプログラムを書き換えたりするなんていかにも泥臭く思える。メモリ上にビットを詰め込んだり、マシンサイクルを余すところなく使い切るような職人技は、話のタネにはなっても、実務に持ち込もうとすると敬遠されかねない。

では、なぜ今、バイナリなのか。

道具の力とは、抽象化の力だ。泥くさい現実を特定の切り口で簡略化することで、本質に関係のない膨大な雑事を気にせずに、考えたい問題のみに集中できる。だが、抽象化にはそれが成り立つ前提が必ずある。システムが正常動作してる間は前提の存在はほとんど気にされることがないが、システムがリミットぎりぎりまで使い込まれて、マージンがなくなってくると、抽象化の壁がほつれてくるのだ。抽象化されたハイレベルな世界でいくら強固なロジックを組み立てても、土台がぐらついたらそのロジックはぺしゃんこだ。それを建て直すには、少なくともほつれた抽象化の壁のひとつ向こうの世界を知ってないとどうにもならない。

ハイレベルな道具を使うだけのプログラマは、抽象化の箱庭の中で遊んでいるようなものだ。その中でも面白いことはできるし、趣味で作っているならこれほど便利なものはない。しかし抽象化の壁に無自覚であることは、自分の世界の限界に無自覚であることだ。プロのプ

ログラマにとっては致命的である。トラブルに対応できないというだけではなく、箱庭の製作者の想定したアイディアを抜けることができないからだ。例えば、関数呼び出しとリターンというイディオムを公理として受け入れ、そのメカニズムを知らなければ、継続渡し形式を使いこなすことは難しいだろう。プログラマとしての底力をつけたければ、本書でしっかり勉強しよう。

と、ここまでは表向きの話。実は、本書の真の魅力は別のところにある。

子供の頃、粗大ごみ置き場に捨てられたテレビを見つけて、ほこりだらけの裏蓋を外してみたことはなかっただろうか。血管のようにうねる色とりどりの配線。ガラス管に封印された不思議な部品。何かいけないものを覗き見ているような興奮。いや、大人になった今でさえ、ウェブのPC系ニュースサイトで最新ノートPCの分解レポートを見つけると、思わず基板写真に見入ってしまわないだろうか。

あなたが私や本書の執筆陣と同じものを持っているはずなら、何でもそろっている最新の開発環境に、満たされない何かを感じているはずだ。ブラックボックスをこじあけて、中を見てみたい。時間さえ許せば、何もないところから自分で組み立ててみたい。

たとえ開発効率が悪くても、泥臭くてスマートじゃなくても、箱庭の外には、どきどきする魅力がある。あなたがプログラミングに関わることを選んだ、その原点が埋もれているからだ。アジャイルだの、ドッグイヤーだのもいいけれど、時には好奇心の向くまま、低レベル世界を探検してみようじゃないか。本書はその絶好のガイドブックになるだろう。

Enjoy Hacking.

川合 史朗

クレジット

著者について

高林 哲　Satoru Takabayashi

ソフトウェアエンジニア。1997 年に全文検索システム「Namazu」を開発。以来、多数のフリーソフトウェアを開発している。平成 16 年度 IPA「未踏ソフトウェア創造事業」において「ソースコード検索エンジン gonzui」を開発、スーパークリエータとして認定される。博士（工学）。趣味はバッドノウハウ。`http://0xcc.net/`

鵜飼 文敏　Fumitoshi Ukai

Debian Project オフィシャルメンバー、元 Debian JP Project リーダー、日本 Linux 協会前会長、The Free Software Initiative of Japan 副理事長、平成 15 年度、16 年度「未踏ソフトウェア創造事業」プロジェクトマネージャー。大学院在籍中に 386BSD や Linux を PC98 アーキテクチャで動かして以来、フリーなオペレーティングシステムの世界にはまる。Debian JP Project 創設時のメンバーで、以後 Debian を中心に活動。`debian.or.jp` および `linux.or.jp` などの運用管理を行っている。

佐藤 祐介　Yusuke Sato

ソフトウェアエンジニア。早稲田大学理工学部卒業後、ソフトウェア作りの修行を積み、現在は某メーカー系企業で情報家電類のセキュリティ脆弱性検査を行っている。日本SELinux ユーザ会（LIDS-JP）、JSSM セキュア OS 研究会、Linux コンソーシアムセキュリティ部会メンバー。

浜地 慎一郎　Shinichiro Hamaji

技術を変な方向、意外な方向に転用するのが好き。雑学好きなので色々作ってみる。しかしたいてい興味が移るので結果としてよくわからないフリーソフトウェアを量産することになる。本業は量子情報の研究室にいる大学院生。

首藤 一幸　Kazuyuki Shudo
エンジニアとして「人には作れないものを作る」をモットーに、Javaスレッド移送システム、Just-in-Timeコンパイラ、オーバレイ構築ツールキットなどのソフトウェアを開発してきた。ウタゴエ(株)取締役 最高技術責任者。博士(情報科学)。技術フェチ。広域分散処理、プログラミング言語処理系、情報セキュリティなどに興味を持つ。http://www.shudo.net/

コントリビュータについて

本書には、Binary Hacker(バイナリアン)から寄せられた数々の熱いHackが収録されています。寄稿していただいたバイナリアンたちのプロフィールを紹介します。

後藤 正徳　Masanori Goto
コンピュータ全般の中でも、特にDebian、GNU C LibraryやLinuxカーネルなどオープンソースソフトウェア開発プロジェクトに関心を持って活動。Debian Projectオフィシャル開発者、YLUG(Yokohama Linux Users Group)発起人。現在、メーカー研究所にてデータストレージ、PC クラスタなどの研究開発に関わる。

中村 実　Minoru Nakamura
命令セットとABIの狭間で生きているリアルバイナリアン。MIPS、SPARC、Alpha用のCコンパイラを作ったり、x86、SPARC、Itanium 2用のJava VMの最適化とデバッグを繰り返したりとローレベルな世界で生きてきた。Java バイトコードを見ると安心する。

中村 孝史　Takashi Nakamura
どっかの組み込み屋。デバイス制御はバイナリアンの知識に入るだろうか。あー、PCのOSはハードウェアを隠蔽してしまうのでよくない。でもOSがなかったらそれはそれでもっと困る。

田中 哲　Akira Tanaka
Rubyにかかわっている人。conservative GCはバイナリアンへの道だと思う。罠かもしれない。

八重樫 剛史　Takeshi Yaegashi
日々是佳境也。

野首 貴嗣　Takatsugu Nokubi
file コマンドをライブラリ化した perl モジュール、File::MMagic のメンテナンスをしている。Namazu、KAKASIなどライブラリでないものを無理やりライブラリ化することを続けてきた。

はじめに

本書のテーマは低レイヤのプログラミング技術です。低レイヤとは「生の」コンピュータに近いことを意味します。

ソフトウェアの世界は抽象化の積み重ねによって進歩してきました。アセンブラはマシン語に対する抽象化であり、C言語はアセンブラに対する抽象化です。そして、C言語の上にはさらに、Cで実装された各種のスクリプト言語が存在します。抽象化は低レイヤの複雑な部分を隠蔽し、より生産性、安全性の高い方法でプログラミングする手段を開発者に提供します。

しかし、低レイヤの技術を完全に忘れてプログラミングできるかというと、そうもいきません。性能をとことん追求したい、信頼性をできるだけ高めたい、ときおり発生する「謎のエラー」を解決したい、といった場面では低いレイヤに下りていく必要に迫られます。残念ながら、抽象化は万全ではないためです。

たとえば、RubyやPerlのスクリプトがセグメンテーションフォルトで異常終了する問題が発生したら、Cのレベルまで下りて原因を探る必要があります。ときには、特殊な問題のために「実行中に自分自身のマシン語のコードを書き換える」といったトリッキーなテクニックが必要になることもあります。低レイヤの技術を知らなければ、このような問題を解決することはできないでしょう。

本書の目的は、そういった場面で使えるたくさんのノウハウ「Binary Hack」を紹介することです。Binary Hackという名称は、0か1、すなわちプログラミングで最も低いレイヤに位置するBinaryという概念に由来します。本書ではBinary Hackを「ソフトウェアの低レイヤの技術を駆使したプログラミングノウハウ」と定義し、基本的なツールの使い方から、セキュアプログラミング、OSやプロセッサの機能を利用した高度なテクニックまで広くカバーします。

従来、このようなノウハウはあまりまとめられることはなく、「知る人ぞ知る」的なところがありました。本書の試みはそういったノウハウを集めて誰にでも使えるようにすることです。本書では実践で役立つHackを中心に取りそろえましたが、中にはあまり役に立たないけどおもしろい、というHackも含まれています。本書を通じて、役立つノウハウを身に付けるとともに、低レイヤ技術の楽しさを知ってもらえればと願っています。

本書で扱うこと、扱わないこと

本書では、Binary Hackに不可欠な基本ツールの使い方から、GCCの拡張機能やOSのシステムコール、インラインアセンブラなどを駆使した高度なテクニックなどの話題を中心に取り扱います。対象プラットフォームはUNIX、とりわけGNU/Linuxにフォーカスしています。WindowsのWin32 APIを用いたBinary Hackはあまり扱いませんが、Cygwinを用いたGNUベースの開発環境では本書で取り上げるHackの多くは適用できるはずです。

本書で必要になる知識と参考文献

本書では読者がUNIXのコマンドライン上で基本的な操作ができることを想定しています。UNIXについては「UNIXプログラミング環境」(Brian W. Kernighan、Rob Pike著、石田晴久監訳、アスキー)などの書籍を参考にしてください。

また、本書ではCやC++といったプログラミング言語を使ったコードを紹介していますが、これらのプログラミング言語を扱うための基本的知識については省略しています。Cについては「プログラミング言語C」(B. W. カーニハン著、D. M. リッチー著、石田晴久訳、共立出版)、C++については「プログラミング言語C++」(Bjarne Stroustrup著、長尾高弘訳、アスキー)などの書籍を参照してください。**[Hack #100]** では多くの参考文献を紹介しています。一部のHackではアセンブリ言語を使用しています。本書ではアセンブリ言語の知識がなくても読めるよう、解説を充実させています。

本書の構成

1章　イントロダクション

Binary Hackのイメージをつかみます。本書で使用されるさまざまな技術用語などについて解説し、さらにBinary Hackの最も基本となるツールの紹介を行います。

2章　オブジェクトファイルHack

実行ファイルや共有ライブラリの正体であるオブジェクトファイルについての理解を深めます。まず、GNU/Linuxなどで用いられているELFについて解説し、さらにライブラリに関するHackを紹介します。オブジェクトファイルHackの基本となるGNU Binutilsの使い方も解説します。

3章　GNUプログラミングHack

GNUの開発環境、すなわちGCC、glibcをはじめとするソフトウェアはさまざまな便利な拡張機能を持っています。本章ではGNU開発環境の力を最大限に引き出すテクニックを取り上げます。

4章　セキュアプログラミングHack

セキュアなプログラムを書くことは、現代で最も重要な課題の1つです。本章ではセキュリティホールを防ぐためのテクニックや、セキュリティホールを見つけ出し、退治するための手法を紹介します。

5章　ランタイムHack

プログラムの実行時にプログラムが自分自身を書き換えたり、自分の状態を調べることができたらおもしろいと思いませんか。本章では実行中のプログラムに対して適用できるさまざまなテクニックを紹介します。

6章　プロファイラ・デバッガHack

本章ではプロファイラを使ってプログラムのボトルネックを調べる方法、およびデバッガの高度な使い方を紹介します。本章ではプロファイラとしてgprof、sysprof、oprofileを、デバッガとしてGDBを取り上げます。

7章　その他のHack

本章では以上の章に分類できなかったHackを扱います。最後のHackでは文献案内として今後のバイナリHackの手引きとなる書籍やWebサイトなどを紹介します。

本書の利用法

本書ははじめから順に読み進めても、目次から面白そうな項目を選んでいきなりそこを読んでもかまいません。もし、バイナリ技術の基礎的な知識を仕入れたいと思っているなら、1章をまず目を通すとよいでしょう。また、プログラミングの経験がまだ浅いなら、各章にある初級Hackから読み始めるのがよいでしょう。

本書での表記

本書で用いている表記は以下の通りです。

等幅(`sample`)

サンプルコード、ファイルの内容、コンソールの出力、変数名、コマンド、その他のコードを示しています。

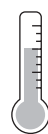
等幅太字(**`sample`**)

ユーザ入力と置き換えられるべきコマンドやテキストを示します。

このアイコンとともに記載されている内容は、ヒント、アドバイス、または一般的な覚え書きです。そのテーマに役立つ補足情報などが記載されています。

このアイコンとともに記載されている内容は、注意または警告を示します。

各 Hack の左隣にある温度計アイコンは、Hack の相対的な難易度を示しています。

初級  中級 上級

サンプルコードの使用について

本書の目的は、読者の作業に役立つ情報を提供することです。一般的には、本書に掲載されているコードを、各自のプログラムまたはドキュメントに使用することができます。コードの大部分を転載する場合を除き、オライリー・ジャパンに許可を求める必要はありません。例として、本書のコードブロックをいくつか使用するプログラムを作成するために、許可を求める必要はありません。なお、オライリー・ジャパンから出版されている書籍のサンプルコードをCD-ROMとして販売したり配布したりする場合には、そのための許可が必要です。本書や本書のサンプルコードを引用して問題に答える場合、許可を求める必要はありません。ただし、本書のサンプルコードのかなりの部分を製品マニュアルに転載するような場合は、そのための許可が必要です。

出典を明記する必要はありませんが、そうしていただければ感謝します。出典を明記する際には、高林哲、鵜飼文敏、佐藤祐介、浜地慎一郎、首藤一幸著『Binary Hacks』(オライリー・ジャパン)のようにタイトル、著者、出版社、ISBN などを盛り込んでください。

サンプルコードの使用について、正規の使用の枠を超える、またはここで許可している範囲を超えると感じる場合には、japan@oreilly.com までご連絡ください。

意見と質問

本書の内容については、最大限の努力をもって検証および確認を行っていますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植などに気づかれることもあるでしょう。本書を読んで気づかれたことがありましたら、今後の版で改善できるようにお知らせください。将来の改訂に関する提案も歓迎します。連絡先を以下に示します。

株式会社オライリー・ジャパン
〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F

電話　03-3356-5227
FAX　03-3356-5261
電子メール　japan@oreilly.com

本書に関する技術的な質問や意見については、次の宛先に電子メールを送ってください。

japan@oreilly.com

本書の Web ページには、正誤表、追加情報が掲載されています。

http://www.oreilly.co.jp/books/4873112885/

オライリーに関するその他の情報については、次の Web サイトを参照してください。

http://www.oreilly.co.jp/
http://www.oreilly.com/

謝辞

共著者の鵜飼文敏さん、佐藤祐介さん、浜地慎一郎さん、首藤一幸さん、およびコントリビュータの後藤正徳さん、中村実さん、中村孝史さん、田中哲さん、八重樫剛史さん、野首貴嗣さんに感謝します。優れたハッカーとともに本書を執筆することができたことは望外の喜びです。

本書への推薦の言葉をいただいた川合史朗さんに感謝します。川合さんは低レベルから高レベル技術まで精通したハッカーであるとともに、優れたライター、翻訳家、俳優という顔も持ちます。川合さんに推薦の言葉をいただいたことはこの上なく光栄です。

本書の発端は2005年末にさかのぼります。当時流行していたWeb 2.0という言葉にかこつけて Binary 2.0 という言葉をブログで提唱したのが 2005 年 11 月、「Binary 2.0 カンファレンス」を開催したのが 2005 年 12 月です。Binary 2.0 の定義は明確ではなく、誰もが何のことかよくわかっていなかったにも関わらず、Binary 2.0 カンファレンスは 100 人を超える参加者でにぎわいました。

そして、イベント会場に遊びに来ていただいたオライリー・ジャパンの渡里さんと田村さんに「Binary Hacks 出しましょう」と話を持ちかけたのが本書の契機となりました。すばやいフットワークで本書の実現の機会を作っていただき、執筆、編集の過程では辛抱強く付き合っていただいた渡里さんと田村さんに感謝します。

2006 年 9 月　高林 哲

HACK #10

objdump でオブジェクトファイルを逆アセンブルする

本Hackでは、objdumpを使ってオブジェクトファイルを逆アセンブルする方法について説明します。

objdump でオブジェクトファイルを逆アセンブルする

objdumpはオブジェクトファイルをダンプするだけではなく、ELFバイナリの場合は逆アセンブルすることができます。逆アセンブルする時は-dオプション(--disassembleオプション)を使います。

```
% objdump -d hello.o

hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 e4 f0                and    $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov    $0x0,%eax
   e:   29 c4                   sub    %eax,%esp
  10:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  17:   e8 fc ff ff ff          call   18 <main+0x18>
  1c:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  23:   e8 fc ff ff ff          call   24 <main+0x24>
```

このように -d オプションで逆アセンブルする時は、通常実行コードがあるセクション(.text など)のみを逆アセンブルの対象とします。すべてのセクションを対象にしたい場合は -D オプション(--disassemble-all オプション)を使います。この場合、.debug_abbrev などコードでない部分もコードだったらどうなるかと解釈して逆アセンブル結果を出力します。

逆アセンブルは通常下のように表示されます。

```
アドレス <シンボル>:
 アドレス: コードのバイト列    逆アセンブルコード
```

コードのバイト列は不要なら、--no-show-raw-insn オプションを使います。

また、--prefix-addressオプションを使うと逆アセンブルコードのアドレスはシンボルからの相対アドレスと共に出力されるようになります。この場合は自動的に--no-show-raw-insnになります。

```
% objdump -d --prefix-address hello.o
```

```
hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:
00000000 <main> push   %ebp
00000001 <main+0x1> mov    %esp,%ebp
00000003 <main+0x3> sub    $0x8,%esp
00000006 <main+0x6> and    $0xfffffff0,%esp
(略)
```

ちなみに--prefix-addressにして、コードのバイト列も見たい場合は--show-raw-insnオプションを同時に使います。

```
% objdump -d --prefix-address --show-raw-insn hello.o

hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:
00000000 <main> 55                              push   %ebp
00000001 <main+0x1> 89 e5                       mov    %esp,%ebp
00000003 <main+0x3> 83 ec 08                    sub    $0x8,%esp
00000006 <main+0x6> 83 e4 f0                    and    $0xfffffff0,%esp
(略)
```

objdumpで特定のセクション、アドレス範囲だけ逆アセンブルする

セクションを指定してそのセクションだけ逆アセンブルすることもできます。セクション指定はダンプする時と同様-jオプション(--sectionオプション)です。

```
% objdump -d -j .init hello

hello:     ファイル形式 elf32-i386

セクション .init の逆アセンブル:

0804829c <_init>:
 804829c:       55                      push   %ebp
 804829d:       89 e5                   mov    %esp,%ebp
 804829f:       83 ec 08                sub    $0x8,%esp
 80482a2:       e8 7d 00 00 00          call   8048324 <call_gmon_start>
 80482a7:       e8 e4 00 00 00          call   8048390 <frame_dummy>
 80482ac:       e8 ff 01 00 00          call   80484b0 <__do_global_ctors_aux>
 80482b1:       c9                      leave
 80482b2:       c3                      ret
```

アドレス範囲もダンプの時と同様--start-addressオプションと--stop-addressオプションで指定できます。

ソースファイルとの対応を表示する

デバッグ情報が含まれているオブジェクトファイルの場合は、-lオプション(--line-numbersオプション)を使うと、それぞれのコードがソースコードのどの行に対応するかという情報も出力してくれます。デバッグ情報が含まれていない場合は-lオプションを指定しても意味がありません。

```
% objdump -d -l hello.o

hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:

00000000 <main>:
main():
/tmp/hello.c:5
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 e4 f0                and    $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov    $0x0,%eax
   e:   29 c4                   sub    %eax,%esp
/tmp/hello.c:6
  10:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  17:   e8 fc ff ff ff          call   18 <main+0x18>
/tmp/hello.c:7
  1c:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  23:   e8 fc ff ff ff          call   24 <main+0x24>
```

さらに-Sオプション(--sourceオプション)を指定すると、もしそのソースファイルがあれば、-lオプションの行番号に対応するソースコードをその場所に挿入して表示してくれるようになります。

```
% objdump -d -S hello.o

hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル:

00000000 <main>:
#include <stdio.h>

int
main(int argc, char *argv[])
{
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
```

```
  6:   83 e4 f0                and    $0xfffffff0,%esp
  9:   b8 00 00 00 00          mov    $0x0,%eax
  e:   29 c4                   sub    %eax,%esp
   printf("Hello, world\n");
 10:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 17:   e8 fc ff ff ff          call   18 <main+0x18>
   exit(0);
 1c:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 23:   e8 fc ff ff ff          call   24 <main+0x24>
```

もちろん-Sオプションと-lオプションを同時に使うこともできます。-Sオプションも-lオプションと同様、オブジェクトファイルにデバッグ情報が含まれていなければ意味がありません。オブジェクトファイルのデバッグ情報としてはソースコードのパス名と行番号が含まれているだけなので、そのソースファイルが、そのパス名で示される場所に置かれている必要があります。そこに対応するソースファイルがなければソースは表示されませんし、違うソースが置かれている場合は異なったソースの行が出力されてしまうことがあります。

リンクする前のオブジェクトファイルでは再配置されるアドレスは0になっていることに注意しましょう。この例では"Hello, world\n"へのポインタは、13～16の4バイトに埋め込まれるはずですが、リンク前なので0のままになっています。

```
c7 04 24 00 00 00 00
```

リンクしてできた実行ファイルで該当する部分には、下のようにアドレスが埋め込まれています。

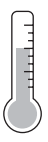
```
c7 04 24 04 85 04 08
```

```
% objdump -d --start-address=0x080483c4 --stop-address=0x80483ec -S hello
(略)
 80483d4:       c7 04 24 04 85 04 08    movl   $0x8048504,(%esp)
(略)
```

まとめ

objdumpを使うことで、オブジェクトファイルや実行ファイルの逆アセンブルを行うことができます。ソースコードが残っていれば対応するソースも逆アセンブルと混合させて出力することもできます。

— Fumitoshi Ukai



HACK #29 ライブラリの外に公開するシンボルを制限する

GNUリンカのバージョンスクリプトおよびGCC拡張を使って、ライブラリの外に公開するシンボルを制限することができます。

C 言語にはファイル内(コンパイル単位)からしかアクセスできない static 関数と、別のファイルからもアクセスできる非static関数があります。しかし、ライブラリを作成する上では、この 2 つのスコープだけでは不十分なときがあります。

本 Hack では、GNU リンカのバージョンスクリプトおよび GCC の拡張を使って、ライブラリの外に公開するシンボルを制限する方法を紹介します。

バージョンスクリプトの場合

GNUリンカのバージョンスクリプトを用いるとライブラリの外に公開するシンボルを制限できます。バージョンスクリプトは名前の通り、シンボルにバージョンをつける用途にも使えます。こちらについては「**[Hack #30]　ライブラリの外に公開するシンボルにバージョンをつけて動作を制御する**」を参照してください。

次のような例を考えてみます。

```
% cat a.c
// foo() は libfoo の主役の関数なので公開したい
void foo() {
  bar();
}

% cat b.c
// bar() はライブラリの中だけで使われるべきなので本当は公開
// したくない。しかし別のファイルに含まれる foo() から使われ
// ているので、非 static にせざるをえない
void bar() {
}
```

このようなコードa.cとb.cをそれぞれコンパイル、リンクしてlibfoo.soを作ると、通常、foo()とbar()の両方の関数のシンボルがライブラリの外に公開されます。しかし、本来bar()は外には公開したくない関数です。

```
% gcc -fPIC -c a.c; gcc -fPIC -c b.c; gcc -shared -o libfoo.so a.o b.o
% nm -D libfoo.so |grep -v '_'
000005e4 T bar
000005d4 T foo
```

そこで、GNUリンカのバージョンスクリプトを用いると、外に公開する関数を制限できます。下の例ではfooをグローバル(ライブラリの外に公開)に、それ以外をローカル(ライブラ

リの中に閉じる）と定義しています。

```
% cat libfoo.map
{
  global: foo;
  local: *;
};
```

このようなバージョンスクリプトlibfoo.mapをgccに-Wl,--version-script,libfoo.mapで渡してリンクすると、barは隠れてfooだけがライブラリの外に公開されます。-Wlはカンマで区切られたパラメータをリンカに渡すというオプションです。

```
% gcc -fPIC -c a.c; gcc -fPIC -c b.c; gcc -shared -o libfoo.so a.o b.o \
  -Wl,--version-script,libfoo.map
% nm -D libfoo.so |grep -v '_'
000004d8 T foo
```

このとき、シンボルbarはリンカによって隠されるだけなので、foo()からbar()への呼び出しはPICコードの流儀に従って、PLTを経由します。つまり、シンボルは隠れても関数呼び出しの方法は変わりません。

メリット

公開するシンボルを制限することには次のようなメリットがあります。

- 非公開APIをライブラリの利用者に見せない
- 共有ライブラリ内のシンボルテーブルを小さくし、動的リンクのコストを軽減する

動的リンクのコストは小さなソフトウェアではほとんど無視できますが、FirefoxやOpenOffice.orgといった巨大なソフトウェアでは大きな問題になります。動的リンクのコストについては「**[Hack #085]　prelinkでプログラムの起動を高速化する**」を参照してください。

C++の場合

C++の場合も、Cの時と大体同じですが、バージョンスクリプトの書き方は少しだけ変わります。以下に例を示します。

```
{
  global:
    extern "C++" {
      some_class::some_func*
    };
  local: *;
}
```

ポイントは、シンボル名のまわりをextern "C++" {}で囲むことと、関数名の後ろに*を付けることです。extern "C++" を付けるとデマングルした形でC++のシンボルをマッチできます。デマングルしたC++のシンボルには引数の型の情報が含まれるため、関数名の後ろに*を付けないとマッチしません。C++のシンボルのデマングルについては「**[Hack #14] c++filtでC++のシンボルをデマングルする**」を参照してください。

GCC拡張の場合

GCC拡張を使って公開するシンボルを制限する方法もあります。最適化という観点では、リンクの時点でシンボルを制限するよりも、コンパイルの時点で行ったほうが効果的です。

具体的にはGCCの__attribute__((visibility("hidden")))および__attribute__((visibility("default")))という属性を使って公開するシンボルの制限を行います。これらの拡張はGCC 4.0以降から利用できます(GCC 3.xからサポートが開始されましたが、C++のクラスに適用できるようになったのは4.0からです)。

それでは例を見てみましょう。次のプログラムではfunc1とクラスFooに対して、シンボルを公開するように明示的に属性を付け、func2()とクラスBarについては何も付けていません。

```
#define EXPORT __attribute__(("default")))

EXPORT void func1() {}
void func2() {}

struct EXPORT Foo {
    void func();
};
void Foo::func() {}

struct Bar {
    void func();
};
void Bar::func() {}
```

これらのプログラムを普通にビルドして出来上がった共有ライブラリを見ると、すべてのシンボルが公開されていることがわかります。

```
% g++ -o test.so  -shared test.cc
% nm --demangle -D test.so |grep func
000006ec T func1()
000006f2 T func2()
000006fe T Bar::func()
000006f8 T Foo::func()
```

一方、-fvisibility=hidden オプションを g++ に渡して、デフォルトの visibility を hidden

にすると、明示的に__attribute__(("default")))で公開されていないシンボルはすべて外に出ないようになります。今回の例ではfunc2()とBarのメンバ関数が隠されました。

```
% g++ -o test.so -fvisibility=hidden -shared visibility.cc
% nm --demangle -D test.so |grep func
000006ac T func1()
000006b8 T Foo::func()
```

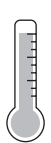
上の例とは逆に__attribute__(("hidden")))を使って明示的にシンボルを隠す方法もありますが、デフォルトですべて隠して公開したいものだけを明示する上の方法の方が意図せずにシンボルが漏れてしまう可能性が低くなります。

バージョンスクリプトを使う方法と比べてvisibility属性を使った方法の方が、高速なコードを生成できます。visibilityがhiddenな関数はPLTを経由せずに直接呼び出せるようになるからです。

まとめ

GNUの開発環境において、ライブラリの外に公開するシンボルを制限する方法を紹介しました。大規模なプロジェクトで共有ライブラリを利用するときに特に役に立つノウハウではないかと思います。

—— Satoru Takabayashi



HACK #73 libunwindでコールチェインを制御する

libunwindを用いると、コールチェインの情報を得ることや、その情報を使ってunwindすることが可能になります。

ここでは、コールチェインを制御するライブラリである、libunwind(http://www.hpl.hp.com/research/linux/libunwind/)を紹介します。libunwindは、HPによって開発されているライブラリであり、MITライセンスで配布されています。現在のところ、libunwindはIA-64 Linuxについて完全にサポートされており、x86 LinuxとIA-64 HP-UXに対しても基本的なサポートがなされています。

一般にunwindとは、スタックの巻戻し処理を意味します。典型的な巻戻しとしてはC言語のreturn文がありますが、libunwindを用いると複数の関数をまたがって一気に巻戻しをすることが可能になります。また、コールチェインの情報を取得できるため、バックトレースやどこから呼ばれたかという情報を手軽に取得することができます。

本Hackでは、libunwindの簡単な機能を紹介します。

libunwind でバックトレースを表示する

以下に libunwind を使ってバックトレースを表示するプログラムを示します。

```
#include <libunwind.h>

void show_backtrace() {
    unw_cursor_t cursor;
    unw_context_t uc;
    unw_word_t ip, sp;
    char buf[4096];
    int offset;

    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    while (unw_step(&cursor) > 0) {
        unw_get_reg(&cursor, UNW_REG_IP, &ip);
        unw_get_reg(&cursor, UNW_REG_SP, &sp);
        unw_get_proc_name(&cursor, buf, 4095, &offset);
        printf("0x%08x <%s+0x%x>\n", (long)ip, buf, offset);
    }
}

void func() {
    show_backtrace();
}

int main() {
    func();
    return 0;
}
```

特に難しい点はないはずです。実行結果は、以下のようになります。

```
% ./a.out
0x080489c9 <func+0xb>
0x080489ec <main+0x21>
0x41032d5f <__libc_start_main+0xdf>
0x0804886d <_start+0x21>
```

libunwind で unwind する

次にlibunwindで複数回returnをする例も見てみます。以下にサンプルコードを示します。

```
#include <libunwind.h>

void skip_func() {
    unw_cursor_t cursor;
    unw_context_t uc;
```

```
    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    unw_step(&cursor);
    unw_step(&cursor);
    unw_resume(&cursor);
    printf("will be skipped.\n");
}

void skipped_func() {
    skip_func();
    printf("will be skipped.\n");
}

int main() {
    printf("start.\n");
    skipped_func();
    printf("end.\n");
    return 0;
}
```

skip_func 内では、unw_step を二度呼んで、skip_func => skipped_func => main とスタックフレームへのカーソルを巻き戻した後、unw_resume で復帰しています。これによって一気に main まで復帰するため、2 つの "will be skipped.\n" は出力されません。

自力で unwind する

環境を限定すれば、自力でunwindすることも難しくはありません。ここでは、getcontext/setcontext(2)を用いて、自力で簡単な unwind をする方法を紹介します。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <ucontext.h>

typedef struct layout {
    struct layout *ebp;
    void *ret;
} layout;

void skip_func() {
    ucontext_t uc;
    layout *ebp = __builtin_frame_address(0);
    ebp = ebp->ebp;

    getcontext(&uc);
    uc.uc_mcontext.gregs[REG_EIP] = (unsigned int)ebp->ret;
    uc.uc_mcontext.gregs[REG_EBP] = (unsigned int)ebp->ebp;
    setcontext(&uc);

    printf("will be skipped.\n");
```

```
}

void skipped_func() {
    skip_func();
    printf("will be skipped.\n");
}

int main() {
    printf("start.\n");
    skipped_func();
    printf("end.\n");
    return 0;
}
```

ここでは、「**[Hack #63]　Cでバックトレースを表示する**」で紹介したスタックフレームをさかのぼる方法を用いて戻った先でのebpとeipを取得しています。getcontext/setcontext(2)で使用しているucontext構造体のuc_mcontextメンバは、「**[Hack #78]　シグナルハンドラからプログラムの文脈を書き換える**」でも解説したように、実装依存となります。文脈の代入部分は環境にあわせて書き換えて下さい。

libunwindでは、ここで紹介した方法を含めて、アーキテクチャごとに有効なさまざまな方法を実行するようになっています。

その他の機能

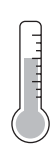
libunwindでは、以上のような処理をptrace(2)ごしに行うことによって、別プロセスのコールチェインの情報を取得したり、操作したりする方法も提供されています。

また、効率的なsetjmp/longjmpも提供されています。これはsetjmpは高速な代わりにlongjmpは低速になっていますが、例外処理などに使用するのであれば適切なトレードオフと言えるでしょう。

まとめ

unwindとは複数関数をまたがってさかのぼることのできるreturnのようなものです。libunwindを用いると、コールチェインの情報を得たり、その情報を使ってunwindすることができます。

—— Shinichiro Hamaji



HACK #77 関数への enter/exit をフックする

GCCの-finstrument-functionsオプションを使うと、関数へのenter/exit時に自作の関数を呼び出すことができます。

GCCの-finstrument-functionsというオプションを利用すると、C/C++の関数が呼び出された直後と、その関数からreturnする直前に、自作の関数を呼び出してもらうことができます。本Hackでは、この-finstrument-functionsの使い方を説明します。

使い方

次のようなフック関数をソースコードのどこかに書き足し、ソースコード全体を-finstrument-functionsオプション付きでコンパイルするだけで、関数のenter/exitをフックすることができます。とてもシンプルです。

```
__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *func_address, void *call_site) {
  // 関数へのenter時に行う処理
}
__attribute__((no_instrument_function))
void __cyg_profile_func_exit(void *func_address, void *call_site) {
  // 関数からのexit時に行う処理
}
```

引数func_addressは、今enterした/今exitしようとしている関数のアドレスです。call_siteは、その関数を呼んだ関数のアドレスです。

活用例：プロセスのスタック使用量を測定する

-finstrument-functionsにはさまざまな使い途が考えられますが、例として「プログラムのスタック使用量を動的に測定する」というのを試してみましょう。

フック関数の作成と共有ライブラリ化

まず、現在のスタック使用量を計算するフック関数__cyg_profile_func_enterを作成します。__cyg_profile_func_exitは、特に行う処理がないので作成しません。

```
// stack_usage.c
#include <stdio.h>
#include <stddef.h>
static ptrdiff_t max_usage = 0;

__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *func_address, void *call_site) {
```

```
    extern void *__libc_stack_end;
    const ptrdiff_t usage = __libc_stack_end - __builtin_frame_address(0);
    if (usage > max_usage) max_usage = usage;
  }

  __attribute__((no_instrument_function, destructor))
  static void print_usage() {
    printf("スタック使用量:約%tdバイト\n", max_usage);
  }
```

print_usageはプログラムの終了時に自動的に実行される関数です。GCCのデストラクタ機能を使用しています。詳しくは「**[Hack #22]　GCCのGNU拡張入門**」や「**[Hack #31]　main()の前に関数を呼ぶ**」を参照してください。ここで、stack_usage.cを共有ライブラリ化しておきます。

```
% gcc -fPIC -shared -o stack_usage.so stack_usage.c
```

測定対象プログラムのmake

測定対象のプログラムとして、gzipを選んでみました。これを-finstrument-functions付きでmakeします。

```
% tar xf gzip-1.2.4a.tar && cd gzip-1.2.4a
% CFLAGS=-finstrument-functions ./configure && make
```

__cyg_profile_func_{enter,exit}は、空の実装がglibcに含まれているため、gzipのソースコードを改変をしなくてもgzipコマンドのリンクは成功します。

測定

先ほどのstack_usage.soをプリロードしてgzipを実行すると、スタック使用量が標準出力に表示されます。

```
% LD_PRELOAD=../stack_usage.so ./gzip sample.txt
スタック使用量: 約716バイト
```

なお、測定対象プログラムがマルチスレッドの場合や、もっと正確な測定を行いたい場合は、stack_usage.cにもう一工夫必要です。詳しくは「**[Hack #75]　スタック領域のアドレスを取得する**」を参考にしてください。また、スタックの使用量を把握する方法として、「**[Hack #66]　プロセスや動的ライブラリがマップされているメモリを把握する**」にあるような/proc/<pid>/mapsを用いる方法もあります。

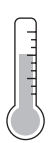
もう1つのフック方法(LD_AUDIT)

バージョン2.4以降のglibcを使っている場合は、LD_AUDITという環境変数を用いることで、gzipなどの測定対象を再コンパイルすることなしに、(PLTを経由した)すべての関数呼び出しを自由にフックすることができます。詳しくは、Sun Solarisの日本語オンラインマニュアル、「実行時リンカーの監査インターフェース」を参照するのが、今のところ便利でしょう。glibc にもほぼそのまま適用できる内容になっています。

まとめ

GCC の -finstrument-functions オプションを使うと、関数への enter/exit 時に自作の関数を呼び出すことができます。この機能を活用すると、プログラムの動作の動的な解析をお手軽に行うことが可能です。

—— Yusuke Sato



HACK #79 プログラムカウンタの値を取得する

x86、SPARC、PowerPCなどでは、プログラムカウンタ(PC)に対して通常のレジスタのようにアクセスすることができません。そういった場合にPCの値を取得する方法を紹介します。

本 Hack では、プログラムカウンタの値を取得する方法を紹介します。

サブルーチン呼び出し命令の応用

バイナリアンはしばしば、プログラムカウンタ(PC)の値を取得する必要に迫られます(「**[Hack #80] 自己書き換えでプログラムの動作を変える**」)。PCの書き換えは簡単です。ジャンプ命令で行うことができます。しかし、値の取得、すなわち汎用レジスタまたはメモリへのコピーは、そのための命令が用意されていないことが多く、一筋縄ではいきません。

ここで、例えばARMプロセッサなら、PCに対しても汎用レジスタと同様にアクセスできるので苦労はありません。しかし他の多くのアーキテクチャ、例えばx86、SPARC、PowerPC、MIPS などはそうはなっていないため、工夫が必要です。

以下に、x86でPCを値として取得するインラインアセンブリコードを示します。このコードを実行すると、変数 p に popl 命令のアドレスが格納されます。

```
void *p;

  asm(".byte 0xe8,0,0,0,0\n\t"   /* call the following popl insn */
      "popl %0\n\t"
      : "=m" (pc));
```

0xe8はサブルーチン呼び出し命令callです。相対アドレスであるところの引数の値が0なので、続くpopl命令を関数だと思い込んで、呼び出します。call命令は、callから戻った後の実行再開アドレス、つまり続くpopl命令のアドレスをスタックに積むということに注意してください。結局、callに続いて実行されるpopl命令は、popl命令自身のアドレスをスタックからポップすることになります。

あるいは、次のように1というラベルを使うこともできます。call 1fは前方(forward)にある1というラベルをcallするという意味です。

```
void *p;
asm("call 1f; 1: popl %0" : "=m"(p));
```

他のアーキテクチャでも同様の方法でPCの値を取得することが可能です。例えばPowerPCではbl命令に続くmflr命令、MIPSではjal命令で、PCの値を汎用レジスタにコピーできます。

まとめ

x86などでは、プログラムカウンタ(PC)に対して通常のレジスタのようにはアクセスできません。そういったプロセッサ上でPCの値を取得する方法を紹介しました。

―― Kazuyuki Shudo

HACK #101

gzip や bzip2 などを区別せずに伸長する

magicを利用して、ファイル名に依存せずに圧縮形式を自動判別して伸長する方法を紹介します。

本Hackでは、「**[Hack #03]　file でファイルの種類をチェックする**」で解説しているmagicの応用として、gzipやbzip2などの圧縮ファイルの判別と伸長を自動的に行う方法を紹介します。

less と LESSOPEN

lessには、入力ファイルを前処理する機能があります。マニュアルにはこの機能の使いかたとして圧縮ファイルを伸長して表示する例が載っています。実際に載っている例は以下のようなスクリプトを用意し、環境変数LESSOPENに"|lesspipe.sh %s"という内容を設定するものです。

lesspipe.sh:

```
#! /bin/sh
case "$1" in
*.Z) uncompress -c $1  2>/dev/null
     ;;
esac
```

そのようにすると、lessに指定したファイル名の拡張子がZであれば、uncompressして表示します。ところが、このファイル名で判別するやりかたでは、実際の圧縮形式を反映した拡張子がついていなければなりません。しかし、magicを使用すればファイル名に依存せずにファイルの圧縮形式を調べることができます。

具体的には次のようなスクリプトをLESSOPENに"|lesspipe %s"として指定します。

lesspipe:

```
#!/usr/bin/ruby

require 'fileutils'

File.open(ARGV[0]) {|f|
  magic = f.read(3) || ''
  case magic
  when /\ABZh/
    IO.popen('bzip2 -dc', 'w') {|gzip|
      gzip.print magic
      FileUtils.copy_stream(f, gzip)
    }
  when /\A\037\213/, /\A\037\235/
    IO.popen('gzip -dc', 'w') {|gzip|
      gzip.print magic
      FileUtils.copy_stream(f, gzip)
    }
  else
    print magic
    FileUtils.copy_stream(f, STDOUT)
  end
}
```

このスクリプトは、まず指定されたファイルから先頭の3バイトを読み込み、そのmagicによってファイル形式を判別しています。

magic	形式
BZh	bzip2
\037\213	gzip
\037\235	compress（gzipで伸長できる）

まとめ

magicを利用すると、ファイル名に依存せずに圧縮形式を自動判別して伸長することができます。

―― Akira Tanaka

HACK #102 mtrace でメモリデバッグ

本 Hack では、メモリ割り当てのバグを見つけるために mtrace を使う方法を説明します。

mtrace とは？

mtraceとはmalloc関係のトレースを取るようにするためのGNU拡張機能であるmtrace(3)およびその出力ファイルを読みやすい形式に直すためのコマンドラインツール mtrace(1)です。

まずメモリ割り当てのデバッグをしたい対象プログラムでmtrace(3)を呼ぶようにしておきます。mtrace(3)を呼び出すことで適切なフック関数がメモリ割り当て関数群(malloc や free など)に設定されます。

mtrace(3)を呼び出しているプログラムを普通に実行しても、mtrace(3)がない時と同様に動作します。環境変数 MALLOC_TRACE にファイル名を指定して対象プログラムを実行するとmtrace(3)の呼び出し以降に、環境変数MALLOC_TRACEで指定したファイルにメモリ割り当てに関するログを出力するようになります。このトレースは muntrace(3)の呼び出しまで行われます。

このようにして出力されるトレースログファイルはASCIIテキストなので、がんばれば読むことができます。しかし人が読みやすいフォーマットではありません。そこでトレースログファイルを人が読みやすいような形式に変換するツールが mtrace(1)です。mtrace(1)に実行バイナリと出力されたトレースログファイルを指定すると、実行バイナリのデバッグ情報を解釈して、ソースファイルのどの行のメモリ割り当て関数の呼び出しが問題かをリストアップしてくれます。実行バイナリにデバッグ情報が含まれていない時はソースファイルとその行番号ではなく、メモリ割り当て関数を呼びだしたメモリアドレスが使われます。

mtrace を使った例

ここで簡単な例を挙げてみます。

```
% cat -n foo.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
```

```
4  #include <mcheck.h>
5
6
7  struct st {
8      int n;
9      char *name;
10 };
11
12 struct st *
13 new_st(char *name)
14 {
15     static int n = 0;
16     struct st *s;
17     s = (struct st *)malloc(sizeof(struct st));
18     s->n = n++;
19     s->name = malloc(strlen(name)+1);
20     strcpy(s->name, name);
21     return s;
22 }
23
24 void
25 free_st(struct st *s)
26 {
27     free(s);
28 }
29
30 int foo(int n, char *tmpl)
31 {
32     int i;
33     char *name;
34     struct st **s;
35     s = (struct st**)malloc(n * sizeof(struct st *));
36     name = (char *)malloc(strlen(tmpl)+8);
37     for (i = 0; i < n; i++) {
38         sprintf(name, "%s%d", tmpl, i);
39         s[i] = new_st(name);
40     }
41     for (i = 0; i < n; i++) {
42         printf("%d: %s\n", s[i]->n, s[i]->name);
43     }
44     for (i = 0; i < n; i++) {
45         free_st(s[i]);
46     }
47     free_st(s[0]);
48 }
49
50 int
51 main(int argc, char *argv[])
52 {
53     int n = atoi(argv[1]);
54     char *tmpl = argv[2];
55     printf("n=%d tmpl=%s\n", n, tmpl);
```

```
56      mtrace();
57      foo(n, tmpl);
58      muntrace();
59      exit(0);
   60  }
%
```

これを普通にコンパイルします。

```
% ls
foo.c
% cc -g foo.c
% ls
a.out* foo.c
% ./a.out 3 foo
n=3 tmpl=foo
0: foo0
1: foo1
2: foo2
% ls
a.out* foo.c
```

このように普通に実行した場合はmtrace(3)自体が何かをするわけではありません。mtrace(3)でトレースログを出力するようにしたい場合は、環境変数MALLOC_TRACEにトレースログのファイル名を指定して実行します。

```
% export MALLOC_TRACE=mtrace.log
% ./a.out 3 foo
n=3 tmpl=foo
0: foo0
1: foo1
2: foo2
% ls
a.out*  foo.c  mtrace.log
%
```

mtrace.log は次のように単なるテキストファイルです。

```
% head mtrace.log
= Start
@ ./a.out:[0x80485e9] + 0x804a378 0xc
@ ./a.out:[0x8048602] + 0x804a388 0xb
@ ./a.out:[0x8048577] + 0x804a398 0x8
@ ./a.out:[0x80485a1] + 0x804a3a8 0x5
@ ./a.out:[0x8048577] + 0x804a3b8 0x8
@ ./a.out:[0x80485a1] + 0x804a3c8 0x5
@ ./a.out:[0x8048577] + 0x804a3d8 0x8
@ ./a.out:[0x80485a1] + 0x804a3e8 0x5
@ ./a.out:[0x80485d3] - 0x804a398
%
```

mtrace(1)を使うと次のようにわかりやすい形式に修正してくれます。

```
% mtrace a.out $MALLOC_TRACE
- 0x0804a398 Free 13 was never alloc'd /tmp/x/foo.c:28

Memory not freed:
-----------------
   Address     Size     Caller
0x0804a378      0xc  at /tmp/x/foo.c:35
0x0804a388      0xb  at /tmp/x/foo.c:36
0x0804a3a8      0x5  at /tmp/x/foo.c:19
0x0804a3c8      0x5  at /tmp/x/foo.c:19
0x0804a3e8      0x5  at /tmp/x/foo.c:19
%
```

この出力から以下のことがわかります。

- foo.cの28行で割り当てられていない領域をfreeしようとした。
- 以下のところでメモリ割り当てしたものが解放されていない。
  - ・foo.cの35行目で0xcバイト(12バイト)を割り当てた
  - ・foo.cの36行目で0xbバイト(11バイト)を割り当てた
  - ・foo.cの19行目で5バイトを3回割り当てた

もし実行バイナリにデバッグ情報がない場合は、このようにソースコードの対応する場所は表示されずメモリアドレスだけになります。

```
% strip a.out
% mtrace a.out $MALLOC_TRACE
- 0x0804a398 Free 13 was never alloc'd 0x80485d3

Memory not freed:
-----------------
   Address     Size     Caller
0x0804a378      0xc  at 0x80485e9
0x0804a388      0xb  at 0x8048602
0x0804a3a8      0x5  at 0x80485a1
0x0804a3c8      0x5  at 0x80485a1
0x0804a3e8      0x5  at 0x80485a1
```

mtraceの仕組み

mtraceはglibcのmalloc_hook(3)を使っています。mtrace(3)で以下のフックをトレース用に置き換え、muntrace(3)でそれを元に戻しています。

```
void *(*__malloc_hook)(size_t size, const void *caller);
void *(*__realloc_hook)(void *ptr, size_t size, const void *caller);
void *(*__memalign_hook)(size_t alignment, size_t size, const void *caller);
void (*__free_hook)(void *ptr, const void *caller);
```

mtrace(3)でおきかえられたトレース関数は以下のような出力を環境変数MALLOC_TRACEで指定したファイルに出力します。

__malloc_hook
　呼び出し元　+　malloc された領域へのポインタ　malloc したサイズ

__realloc_hook
　失敗した場合
　　呼び出し元　!　realloc しようとしたポインタ　realloc するサイズ
　NULL ポインタを realloc した場合
　　呼び出し元　+　realloc で割り当てられた領域へのポインタ　realloc したサイズ
　成功した場合
　　呼び出し元　<　realloc しようとしたポインタ
　　呼び出し元　>　realloc で割り当てられた領域へのポインタ　realloc したサイズ

__memalign_hook
　呼び出し元　+　memalignで割り当てられた領域へのポインタ　memalignしたサイズ

__free_hook
　呼び出し元　-　free しようとしたポインタ

mtrace の注意点

mtraceは非常に単純にメモリ割り当て関数が呼ばれた時にその呼びだし元とサイズを記録しているだけです。したがってこれですべてのメモリ割り当てバグがとれるわけではありません。

例えば 36 行目を次のように変更したとします。

```
36      name = (char *)malloc(2);
```

これで同じように実行すると明らかに38行目のsprintf(3)でオーバーフローしてしまいます(2 バイトしか割り当てていない領域に 5 バイト書きこみのが理由)が、そのようなことは mtrace では報告されません。

また、19行目のメモリ割り当てを解放するために、27行目を次のように変更したとします。

```
27      free(s); free(s->name);
```

これは明らかにfree(3)する順番が間違っていますが、mtraceではそのような報告はされません。

まとめ

メモリ割り当てのバグをみつけるためのmtraceについて説明しました。

ただし、mtraceではあまり実用的とはいえず、実際にはValgrindなどを利用したほうがよいでしょう。「**[Hack #54]　Valgrindでメモリリークを検出する**」や「**[Hack #55]　Valgrindでメモリの不正アクセスを検出する**」を参照してください。

―― Fumitoshi Ukai