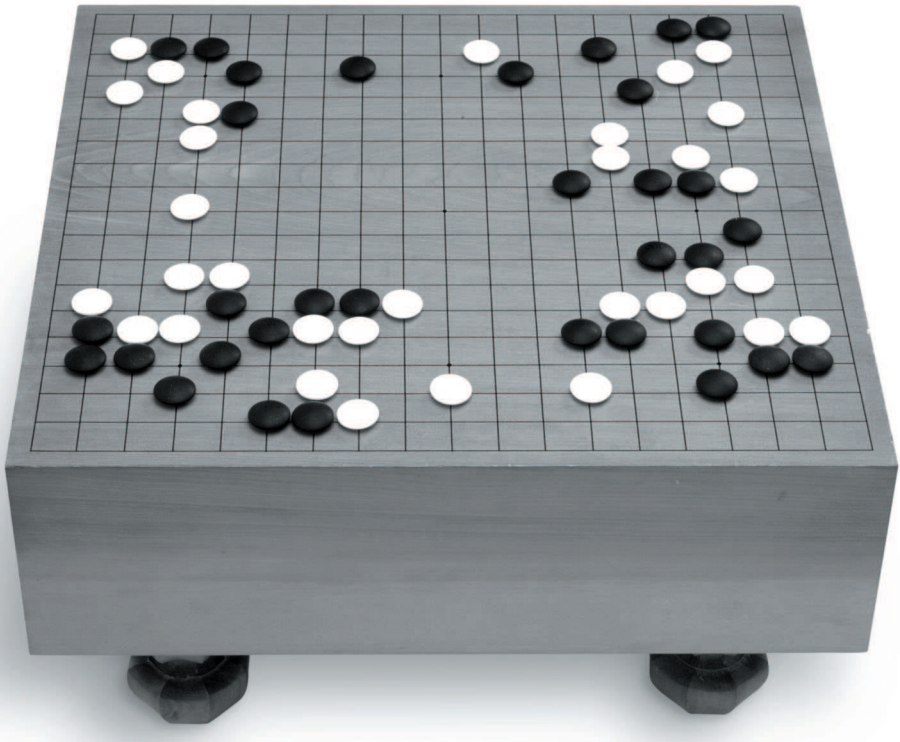
BINARY HACKS™

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

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.co.jp

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.co.jp

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

目　次

1章 イントロダクション
Hack #1-4

HACK #1

Binary Hack 入門

Binary Hackで用いられているテクニックを「各種ツール・ライブラリ」「バイナリフォーマット」「システムコール」「OS 固有の機能」「プロセッサの機能」「コンパイラ固有の機能」に分類して紹介します。

Binary Hack の世界へようこそ。一口に Binary Hack といっても、各種ツールを活用するものや、コンパイラの拡張機能を駆使するものなど、いろいろあります。ここでは、どのような種類のテクニックが Binary Hack に用いられているか分類して見ていきます。

各種ツール・ライブラリ

Binary Hack に役立つツール・ライブラリは世の中にたくさん存在します。なかでも重要なのはGNUプロジェクトが提供するGNU Binutilsに含まれる各種ツール・ライブラリです。2章の「オブジェクトファイル Hack」の大半はこの GNU Binutils を使った Hack で占められています。他にも、「**[Hack #3] fileでファイルの種類をチェックする**」、「**[Hack #54] Valgrindでメモリリークを検出する**」、「**[Hack #73] libunwind でコールチェインを制御する**」、「**[Hack #82] straceでシステムコールをトレースする**」、「**[Hack #83] ltraceで共有ライブラリの関数呼び出しをトレースする**」、「**[Hack #89] oprofile で詳細なシステムプロファイルを得る**」、「**[Hack #95] Portable Coroutine Library（PCL）で軽量な並行処理を行う**」、といったHackでBinary Hack に役立つツールとライブラリを紹介します。本書では、単にツール、ライブラリの使い方を紹介するだけでなく、仕組みについても紹介していきます。先人の知恵と成果を生かして Binary Hack をより生産的なものにしましょう。

バイナリフォーマット

実行ファイルやライブラリは構造を持ったバイナリファイルです。このようなバイナリファイルのフォーマットにはさまざまなものがありますが、GNU/Linux では主に ELF というフォーマットが用いられています。バイナリがどのような構造、情報を持っていて、どの

ようにそれを引き出すかを理解することはBinary Hackを行う上で大変役に立ちます。本書では、「**[Hack #8] readelfでELFファイルの情報を表示する**」、「**[Hack #9] objdumpでオブジェクトファイルをダンプする**」、「**[Hack #12] nmでオブジェクトファイルに含まれるシンボルをチェックする**」、「**[Hack #25] glibcを使わないでHello Worldを書く**」、「**[Hack #67] libbfdでシンボルの一覧を取得する**」、「**[Hack #75] オブジェクトファイルを自力でロードする**」、といったHackでバイナリに含まれる情報を活用するテクニックを紹介します。

システムコール

多くのアプリケーションプログラムはOSのシステムコールを直接呼び出さずに書くことができます。Cライブラリにはシステムコールを抽象化した便利な関数が多く含まれており、readやwriteシステムコールの代わりにfread()やfwrite()関数を使えば、システムコールを意識せずにファイル入出力を行うことができます。しかしながら、メモリ関連の高度な処理をしたり、シグナルを処理するといった場面では、システムコールの呼び出しが必要となります。本書では、「**[Hack #34] ヒープ上に置いたコードを実行する**」、「**[Hack #76] sigaltstackでスタックオーバーフローに対処する**」、「**[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**」、「**[Hack #81] SIGSEGVを使ってアドレスの有効性を確認する**」、といったHackでシステムコールを使った各種のテクニックを紹介します。また、「**[Hack #59] システムコールはどのように呼び出されるか**」ではGNU/Linuxシステムにおいてどのようにシステムコールが呼び出されるかを解説します。

OS固有の機能

通常、システムコールやライブラリ関数はPOSIXやANSI Cなどの規格によってAPIが決められており、それらのAPIを守っている限りにおいて、プログラムは別のOSに移植することができます。しかしながら、ポータブルではないOS固有の機能を用いることにより、「普通」のプログラムでは真似できない込み入った処理を実現することも可能です。本書では、「**[Hack #60] LD_PRELOADで共有ライブラリを差し換える**」、「**[Hack #64] 実行中のプロセスのパス名をチェックする**」、「**[Hack #65] ロードしている共有ライブラリをチェックする**」、「**[Hack #75] スタック領域のアドレスを取得する**」、といったHackでOS固有のテクニックを紹介します。

プロセッサ固有の機能

通常、Cなどの高級言語を使っていればプロセッサ間の差異はほとんど吸収されます。しかし、各プロセッサが持つ固有のレジスタや命令を使うにはアセンブラのレベルに降りてい

かなければなりません。本書では「**[Hack #79] プログラムカウンタの値を取得する**」、「**[Hack #91] ハードウェアのデバッグ機能を使う**」、「**[Hack #92] C のプログラムの中でブレークポイントを設定する**」、「**[Hack #96] CPUのクロック数をカウントする**」、「**[Hack #98] x86が持つ浮動小数点演算命令の特殊性**」、といったHackでプロセッサ固有のテクニックを紹介します。プロセッサ固有の機能を使って、プロセッサの性能をフルに発揮させましょう。

コンパイラ固有の機能

コンパイラはCやC++といった言語でプログラムを開発する上で欠かせないツールです。そして、コンパイラはソースコードをマシン語に翻訳するだけでなく、セキュリティの強化や最適化の補助などの機能も多く提供しています。本書では、「**[Hack #22] GCC の GNU 拡張入門**」、「**[Hack #23] GCC で inline アセンブラを使う**」、「**[Hack #24] GCC のビルトイン関数による最適化を活用する**」、「**[Hack #42] GCC セキュアプログラミング入門**」、「**[Hack #46] -fstack-protector でスタックを保護する**」、といったHackでGCC固有の機能を取り上げます。また、「**[Hack #32] GCC が生成したコードによる実行時コード生成**」、「**[Hack #38] g++ の例外処理を理解する(throw)編**」、「**[Hack #41] g++例外処理のコストを理解する**」、といったHackでは、GCCがどのようなコードを生成するかについて解説します。GCCの機能を使ったHackは本書の中でもかなりの量を占めています。

まとめ

ここではBinary Hackで用いられているテクニックを「各種ツール・ライブラリ」「バイナリフォーマット」「システムコール」「OS固有の機能」「プロセッサの機能」「コンパイラ固有の機能」の6つに分類して紹介しました。それでは以降の章で個々のHackを詳しく見ていくことにしましょう。

—— Satoru Takabayashi

HACK #2 Binary Hack 用語の基礎知識

本Hackではは Binary Hacks に登場する用語を紹介します。

Binary Hackの世界には多くの専門用語が存在します。ここでは本書に頻繁に登場する用語のうち代表的なものを簡単に解説します。

ABI(Application Binary Interface)

アプリケーションが守るべきバイナリレベルでの規約集。関数を呼びだす時のスタックやレジスタの使い方や、シンボルの名前マングルのルールなどが決められている。

OS やプロセッサごとに規定される。

API（Application Programming Interface）
アプリケーションプログラムから OS やライブラリの機能を利用するための関数やデータ構造の規約。API を利用したプログラミングは、同じ API をサポートしたプラットフォーム間でソースコードの互換性があるなどの利点が得られる。

BSS セグメント（Block Started by Symbol Segment）
初期化されていないデータが置かれるセグメント。C のグローバル変数で `int global;` のように初期値が設定されていないものなどが入る。オブジェクトファイル内ではサイズを持たず、プログラムの開始時にカーネルによって `0` に初期化される。ELF での名称は `.bss`。

DSO（Dynamic Shared Object）
GNU/Linux では動的リンクの共有ライブラリを DSO と呼ぶことが多い。`.so` という拡張子を持つ。

DWARF（Debug With Arbitrary Record Format）
デバッグ情報を格納するためのデータフォーマット。GNU/Linux+GCC の環境で標準的に用いられている。DWARF および ELF という名前は指輪物語との関係が疑われる。

ELF（Executable and Linking Format）
実行ファイル、オブジェクトファイル、共有ライブラリ、コアファイルに使われるファイルフォーマットの 1 つ。GNU/Linux や FreeBSD などで採用されている。

GCC（GNU Compiler Collection）
GNU の各種コンパイラの一式。元々は GNU C Compiler を意味していた。GNU C Compiler を表す場合は gcc と表記する。

glibc（GNU C Library）
GNU の C ライブラリ。GNU/Linux や Hurd といった OS で用いられている。

GNU（GNU's Not Unix）
本来は GNU プロジェクトが開発している OS のことを指すが、GNU プロジェクトのことを省略して GNU と呼ぶことが多い。

GNU/Linux
Linux カーネルベースのシステムのこと。GNU を頭につけるのはカーネル以外のコ

ンポーネントを開発している GNU プロジェクトへの敬意を表すため。

GOT（Global Offset Table）
PICを実現するために必要なデータ。PICではグローバルなデータのアクセスをGOTを用いて間接参照で行う。

LLP64
long long とポインタがともに 64 ビットの環境。int と long は 32 ビット。64 ビットの Windows では LLP64。

LP64
long とポインタがともに 64 ビットの環境。int は 32 ビット。64 ビット Linux では LP64。多くの Unix 系の OS は LP64 を採用している。

PIC（Position Independent Code）
任意のアドレスにロード可能なコード。データのアクセスやジャンプは相対アドレスで行われる。位置独立コード。

PIE（Position Independent Executable）
位置独立な実行ファイルのこと。モダンなGNU/Linuxで作成できる。セキュリティの向上などのメリットがある。

PLT（Procedure Linkage Table）
動的リンクを実現するために必要なデータ。GOTとともに用い、動的リンクの共有ライブラリの関数の呼び出しを間接化する。

POSIX（Portable Operating System Interface for UNIX）
システムコールやシグナルといった OS の API を定める規格。多くの Unix 系の OS は POSIX に準拠している（または準拠を目指している）。

SUS（Single UNIX Specification）
Unix と名乗る OS のための規格。最新版の SUSv3 はウェブ上で閲覧可能。歴史的経緯はあるが、SUSv3 は POSIX を含んでいる。

TLS（Thread Local Storage）
どのスレッドも同じ名前の変数を使いながらも、実際に格納される値はスレッドごとに独立して保持できる領域のこと。GCCでは、__threadというキーワードを用いて、TLS を使う。

prelink

動的リンクを高速化する手法の1つ。実行ファイルと共有ライブラリを書き換えて、動的リンクのコストの大半を削減する。多くのLinuxディストリビューションで採用されている。

x86

Intel 社の 8086 系のプロセッサの略称。80486 以降は Pentium や Xeon などといった製品名を持つ。IA-32 とも呼ぶ。

x86_32

x86_64 と区別するため、32 ビットの x86 アーキテクチャを x86_32 と表記する場合がある。

x86_64

AMD が設計した x86 上位互換の 64 ビットプロセッサのアーキテクチャ。AMD64 とも呼ばれる。Intel も同じアーキテクチャを採用している。EM64T は Intel による実装の名称。

インラインアセンブリコード(inline assembly code)

Cなどの高級言語のプログラムに埋め込まれたアセンブリ言語のコードのこと。アーキテクチャ依存の処理や最適化などに使われる。

エンディアン

複数バイトにわたるデータをどの順番で格納するかのルール。バイトオーダーとも呼ぶ。エンディアンという呼称はガリバー旅行記に由来する。

オブジェクトファイル(object file)

コンパイラが生成する中間ファイル。実行ファイルやライブラリはオブジェクトファイルをリンクして作られる。GNU/Linuxでは.oという拡張子を持つ。広義に、実行ファイルやライブラリも含めてオブジェクトファイルと呼ぶこともある。

逆アセンブル(disassemble)

マシン語をアセンブリ言語に変換すること。

共有ライブラリ(shared library)

プログラムの実行時にメモリ上で複数のプログラムによって共有されるライブラリ。通常、共有ライブラリは動的リンクされる。静的リンクの共有ライブラリもあるが、まれ。本書では共有ライブラリと呼んだ場合、動的リンクの共有ライブラリのことを指す。共有オブジェクトとも呼ぶ。

再配置（relocation）
　マシン語のコードに含まれるアドレスをリンク時またはロード時に書き換えること。

シグナル（signal）
　プロセスに送られる同期的、非同期的なイベント。POSIXではSIGKILLとSIGSTOP以外のシグナルはシグナルハンドラで処理できる。

シグナルハンドラ（signal handler）
　シグナルを処理する関数。sigaction()またはsignal()関数で設定できる。

シグネチャ（signature）
　名前、および、引数と戻り値の型によって決まる関数の型。通常、CやC++などのコンパイル型言語では名前が同じでもシグネチャが異なる関数を呼び出そうとすると警告やエラーとなる。

システムコール（system call）
　ユーザレベルのアプリケーションからOSカーネルの機能を呼び出すための仕組み。例：read()、fork()

シンボル（symbol）
　一般的には記号を意味するが、Binary Hacksの文脈では、リンカが関数や変数を識別するときに用いる名前のことを指す。

シンボルテーブル（symbol table）
　オブジェクトファイルなどに含まれるシンボルの表。明示的に削除しない限り、実行ファイルやライブラリにも残っている。

スタック（stack）
　スタックフレームを積み上げていくメモリ領域。Binary Hackの文脈では、スタックは「データ構造のスタック」ではなく、「メモリ領域のスタック」を指すことが多い。

スタックフレーム（stack frame）
　引数やローカル変数、保存したレジスタ、戻り値アドレスなど、関数の呼び出しに必要な情報をまとめたもの。略してフレームとも呼ばれる。

スタックポインタ（stack pointer）
　スタックフレームを操作するために用いられるポインタ。x86などはスタックポインタのための専用のレジスタを持っている。RISCプロセッサでは汎用レジスタの中の1本をスタックポインタと決めて使用することが多い。

スレッド(thread)
プログラムの実行の単位の1つ。プロセスとの主な違いはリソース共有の方法にある。通常、スレッド間の方がプロセス間よりもリソースの共有がしやすい。1つのプロセス内に複数のスレッドを持てる。

スレッドセーフ(thread safe)
マルチスレッドプログラムで安全に実行できること。多くの場合、static変数を内部に持った関数はスレッドセーフではない。

セグメンテーションフォルト(segmentation fault)
アクセス不能なアドレスにアクセスしたり、書き込み不能なアドレスに書き込みを行ったときに発生するエラー。C、C++プログラマは頻繁に遭遇する。セグメント違反とも呼ぶ。

実行ファイル(executable file)
実行が可能なファイル。実行可能ファイルとも呼ぶ。GNU/Linuxでは/usr/binなどに入っている。

静的ライブラリ(static library)
静的リンクされるライブラリ。GNU/Linux では .a という拡張子を持つ。

静的リンク(static link)
実行ファイルの生成時にライブラリをリンクすること。通常、ライブラリの内容は実行ファイルの中に取り込まれるため、実行時にはライブラリのファイルは不要。

ツールチェーン(toolchain)
コンパイラ、リンカ、アセンブラなど、ネイティブなプログラムを生成するために必要な一連のツールの総称。

データセグメント(data segment)
初期化されているデータが置かれるセグメント。ELF での名称は .data。

テキストセグメント(text segment)
マシン語のコードが置かれるセグメント。通常、リードオンリーに設定される。ELF での名称は .text。

デバッガ(debugger)
プログラムのバグの原因を調べるのに役立つツール。デバッガ上でプログラムを動かすことによりバックトレースや変数の調査などが行える。

デバッグ情報（debug information）
　デバッガが必要とする情報。実行ファイルや共有ライブラリに埋め込まれている。gcc では -g オプションを付けると生成される。

デマングル（demangle）
　名前マングルされたシンボルを元の読みやすいシンボルに復元すること。例：_ZN3Foo3BarE => Foo::Bar。

動的リンク（dynamic link）
　実行時にライブラリをリンクすること。実行時にライブラリのファイルが必要。ライブラリが存在しないと実行時にエラーが起きる。

動的リンクライブラリ（dynamic link library）
　動的リンクされるライブラリ。WindowsではDLLと呼ばれる。GNU/LinuxではDSOと呼ばれることが多い。

名前マングル（name mangling）
　関数名とシグネチャから一意のシンボルを作成すること。C++ や Java などの言語で用いられる。例：Foo::Bar=>_ZN3Foo3BarE。

バイナリアン（binarian）
　Binary Hack 的な技術に精通したエンジニア。バイナリ者とも。

バックトレース（backtrace）
　現在の関数に至るまでに通った関数の一覧。スタックトレースとも呼ばれる。

ヒープ（heap）
　malloc()などによって動的に確保されるメモリのための領域。Binary Hackの文脈では、ヒープは「データ構造のヒープ」ではなく、「メモリ領域のヒープ」を指すことが多い。自由記憶領域とも呼ばれる。

ブレークポイント（break point）
　デバッガ上でプログラムの実行を一時停止させる個所。関数名やソースコードの行数などによって指定する。

プログラムカウンタ（program counter）
　CPU の中のレジスタの1つで、現在実行している命令のアドレスを保持しているもの。PC と略されることも多い。インストラクションポインタともいう。

プロセス(process)
　プログラムの実行単位の1つ。実行中のプログラムのインスタンス。通常、プロセスは一意のプロセス ID を持つ。

プロファイラ(profiler)
　プログラムのパフォーマンスを解析するツール。本書ではgprof、sysprof、oprofileを紹介する。

呼び出し規約(calling convention)
　関数を呼ぶときにデータをどのようにスタックに積むかといったことの規約。ABIの一種。OS やプロセッサによって異なる。

ランタイム(runtime)
　実行時の意。実行時に発生するエラーのことをランタイムエラーと呼ぶ。

リンク(link)
　オブジェクトファイルやライブラリを結びつけること。再配置などの処理を行う。

リフレクション(reflection)
　実行中の自分自身のプログラムについての情報を調べたり書き換えたりすること。C言語ではリフレクションのための機能は提供されていないが、Binary Hackを用いることによりリフレクションに近い機能を実現できる。

ロード(load)
　実行ファイルやライブラリをメモリ上に配置すること。

まとめ

本Hackでは Binary Hacksに登場する用語を紹介しました。それぞれの用語は本文中でも必要に応じて解説していきます。

—— Satoru Takabayashi

HACK #3 fileでファイルの種類をチェックする

fileコマンドを用いることによって、ファイルの内容からその種類を調べることができます。

fileコマンドを使うことで、任意のファイルが何であるかを調べることができます。ファイルの種類を調べるためには、ファイルの種類に応じて拡張子をあらかじめ決めておき、ファイル名からファイルの種類を推測するという方法が一般的にとられていますが、fileコ

マンドはファイルの内容を読み込み、特徴的なデータ列(シグネチャ)を探し出して種類を特定します。

以下は file コマンドファイル自身を GNU/Linux 上で調べた例です。-i オプションを付けると MIME メディアタイプ文字列で表示されます。

```
$ file /usr/bin/file
/usr/bin/file: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/
Linux 2.2.0, dynamically linked (uses shared libs), stripped
$ file -i /usr/bin/file
/usr/bin/file: application/x-executable, for GNU/Linux 2.2.0, dynamically linked
(uses shared libs), stripped
```

file コマンドは、以下の順序に基づいてファイルの種別を判断します。

- デバイス、ディレクトリ、シンボリックリンクなどのスペシャルファイルチェック
- 圧縮ファイルのチェック
- tar ファイルのチェック
- magic データベースファイルに基づくチェック
- ASCII、Unicode などのテキストファイル種別チェック

上記すべてに当てはまらなければ、単なるバイナリ列と判断されます。

magic データベースは、シグネチャ情報が記録されたものです。通常、/etc/magic、/usr/share/misc/file/magic などの場所に保存されています。

前述の例では、以下のシグネチャ情報が利用されています。

```
#------------------------------------------------------------------------------
# elf:  file(1) magic for ELF executables
#
# We have to check the byte order flag to see what byte order all the
# other stuff in the header is in.
#
# What're the correct byte orders for the nCUBE and the Fujitsu VPP500?
#
# updated by Daniel Quinlan (quinlan@yggdrasil.com)
0       string          \177ELF         ELF
>4      byte            0               invalid class
>4      byte            1               32-bit
(略)
>5      byte            1               LSB
(略)
>>16    leshort         2               executable,
```

magic ファイルのエントリーは 4 つのフィールドからなります。

- 先頭、もしくは前のレベルからのオフセット値
- データの種別
- 値
- 出力する文字列

レベルというのは、オフセット値の前に書かれている ">" を意味します。最初のエントリにマッチしたあと、次の行に ">10" と書かれていれば、さらに先頭から 10 バイト先を読み込んで比較をします。同じ数の">"が並んでいる場合は、順次同様に前のレベルからのオフセット値に対応する内容を見て行きます。マッチしたエントリの先に ">" が 1 つ多いエントリがあれば、続けてそのエントリに対するマッチ処理を行います。

たとえば前述の file 自身を判別する例は、以下のような順序で評価されています。

- 先頭から 0 バイト目が "\177ELF" という文字列とマッチするか確認(1 行目)
- マッチしたので "ELF" と表示
- 先頭から 4 バイト目の 1 バイトが 0 か確認(2 行目)
- マッチしないので次の行へ
- 先頭から 4 バイト目の 1 バイトが 1 か確認(3 行目)
- マッチしたので "32-bit" と表示
- 以下しばらくマッチしないので先へ
- 先頭から 5 バイト目が 1 か確認
- マッチしたので "LSB" と表示
- 以下しばらくマッチしないので先へ
- 先頭から 16 バイト目が 2 か確認
- マッチしたので "executable," と表示
- これを最後まで繰り返す

なお、GNU/Linux システムで広く使われている file コマンドには ELF ファイルを特別に処理するコードが含まれており、「for GNU/Linux 2.2.0」以降の部分は、この特別処理のコードによって出力されています。

まとめ

fileコマンドを用いることによって、ファイルの内容からその種類を調べることができます。また、新しいファイルフォーマットが現れても、そのファイルに固有なシグネチャがわかればmagicファイルにエントリを追加することでそのファイルを認識できるようになります。

逆に、もし新しいバイナリファイルフォーマットを決める必要がある場合には、fileコマンドで調べることができるようなシグネチャをフォーマットに盛り込んでおくとよいでしょう。

magic の書式に関する詳細は、man magic を参照してください。

—— Takatsugu Nokubi

HACK #4 od でバイナリファイルをダンプする

本 Hack では、バイナリファイルをダンプするツール、od の使い方について説明します。

8 進ダンプ

od は octal dump（「octal」は 8 進法という意味）というくらいで、デフォルトではバイナリファイルを 8 進数でダンプして出力します。

```
% od /etc/ld.so.cache | head -5
0000000 062154 071456 026557 027061 027067 000060 001430 000000
0000020 000003 000000 047440 000000 047452 000000 000003 000000
0000040 047475 000000 047505 000000 000003 000000 047526 000000
0000060 047547 000000 000003 000000 047601 000000 047616 000000
0000100 000003 000000 047644 000000 047662 000000 000003 000000
```

行頭の最初のカラムの数字は先頭からのオフセットを 8 進数で表現したものです。1 行ごとに 2 バイトずつ（いわゆる short）を 8 個、つまり 16 バイトずつ出力しています。2 バイト（short）はマシンのバイトオーダーで8進数で表現しています。この場合、最初は、8進数では 062154 ですから、10 進数では 25708、16 進数では 646C、2 進数では 0110010001101100 となります。x86 のようなリトルエンディアンの場合、最初の 2 バイトは 0x6C（01101100）と 0x64（11000100）という意味です。

出力フォーマットを指定する

一般的には8進数で出力されてもわかりにくいでしょう。通常は、バイトごとに16進数などになっているほうがわかりやすいと思います。od では出力フォーマットを -t オプション（--format オプション）で指定することができます。-t オプションには次のような型を指定します。

型	意味
a	文字の名前（7bit ASCII）
c	ASCII 文字かエスケープ文字
d	符号付き 10 進数

型	意味
f	浮動小数点数
o	8進数
u	符号なし10進数
x	16進数

a、cは常にバイト単位の出力になります。d、o、u、xについては、その後ろにまとめて表示するバイト数、または次のようなサイズ指定子を使うことができます。

型	意味
C	char
S	short
I	int
L	long

fについては次のサイズ指定子を使うことができます。

型	意味
F	float
D	double
L	long double

さらにzを付けると右にASCII文字表示も付けることができます。

オフセットの表記も8進数以外にすることもできます。-Aオプションに次のいずれかを指定することでオフセットの基数を変更することができます。

型	意味
d	10進数
o	8進数(デフォルト)
x	16進数
n	オフセットを表示しない

よく使うのが、バイトごとに16進数でダンプすることでしょう。その場合は-t x1 -A xのように指定します。

```
% od -t x1 -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00
```

```
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00
000030 67 4f 00 00 03 00 00 00 81 4f 00 00 8e 4f 00 00
000040 03 00 00 00 a4 4f 00 00 b2 4f 00 00 03 00 00 00
```

ASCII 文字表示も調べたい場合は次のように -t x1z と z を付けます。

```
% od -t x1z -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00  >ld.so-1.7.0.....<
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00  >.... O..*O......<
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00  >=O..EO......VO..<
000030 67 4f 00 00 03 00 00 00 81 4f 00 00 8e 4f 00 00  >gO.......O...O..<
000040 03 00 00 00 a4 4f 00 00 b2 4f 00 00 03 00 00 00  >.....O...O......<
```

c も ASCII 文字表示ですが、これを指定した場合、別の行に分けて表示されるようになります。

```
% od -t x1c -A x /etc/ld.so.cache | head -5
000000 6c 64 2e 73 6f 2d 31 2e 37 2e 30 00 18 03 00 00
     l   d   .   s   o   -   1   .   7   .   0  \0 030 003  \0  \0
000010 03 00 00 00 20 4f 00 00 2a 4f 00 00 03 00 00 00
   003  \0  \0  \0       O  \0  \0   *   O  \0  \0 003  \0  \0  \0
000020 3d 4f 00 00 45 4f 00 00 03 00 00 00 56 4f 00 00
```

ダンプを省略しない

od はデフォルトで、複数行同じ内容がある時、その部分のダンプを省略します。

```
% od -tx1 -Ax /usr/share/apache/icons/deb.png | sed -ne '45,49p'
0002c0 fe f8 fd ff f9 ff ff fb fd ff fb ff fd fd fd fe
0002d0 fd fd ff fd fe fe fe fe 00 00 00 00 00 00 00 00
0002e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000320 00 00 00 00 00 00 00 00 00 00 7a 6c 49 5c 00 00 00
```

このように 0x0002e0 ～ 0x000320 までは 00 がずっと続いているので「*」のように省略されてしまいます。省略しないようにするためには -v オプション（--output-duplicates オプション）を使います。

```
% od -tx1 -Ax -v /usr/share/apache/icons/deb.png | sed -ne '45,49p'
0002c0 fe f8 fd ff f9 ff ff fb fd ff fb ff fd fd fd fe
0002d0 fd fd ff fd fe fe fe fe 00 00 00 00 00 00 00 00
0002e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

文字列

odにはstringsのように文字列ダンプの機能もあります。-sオプション(--stringsオプション)を使うと、最低3文字のASCII文字が続いて\0で終了しているようなバイト列を探して、そのオフセットと内容を表示します。

```
% od -Ax -s /etc/ld.so.cache|head -5
000000 ld.so-1.7.0
007450 libz.so.1
00745a /usr/lib/libz.so.1
00746d libz.so
007475 /usr/lib/libz.so
```

オフセットを出力しなければ(-A n)、strings と同じような出力が得られます。

```
% od -An -s /etc/ld.so.cache|head -5
ld.so-1.7.0
libz.so.1
/usr/lib/libz.so.1
libz.so
/usr/lib/libz.so
```

ただし、微妙に strings の出力とは異なり、文字列はASCII文字の連続に\0で終了しているものだけになります。stringsの場合、ASCII文字が連続さえしていれば、それが\0で終了していなくても出力します。

```
% diff -u =(od -An -s /etc/ld.so.cache) =(strings /etc/ld.so.cache ) | head
--- /tmp/zsh2QzQnw      2006-02-21 01:22:08.657320876 +0900
+++ /tmp/zshesbFEZ      2006-02-21 01:22:08.666319531 +0900
@@ -1,4 +1,5 @@
 ld.so-1.7.0
+glibc-ld.so.cache1.1J
 libz.so.1
 /usr/lib/libz.so.1
 libz.so
```

ダンプしてみると「glibc-ld.so.cache1.1J」はその後には\03があり、\0で終わってはいません。

```
% od -Ax -tx1z /etc/ld.so.cache|sed -ne '596,597p'
002530 67 6c 69 62 63 2d 6c 64 2e 73 6f 2e 63 61 63 68  >glibc-ld.so.cach<
002540 65 31 2e 31 4a 03 00 00 96 89 00 00 00 00 00 00  >e1.1J..........<
```

strings のように文字列の最小の長さを指定することもできます。

```
% od -Ax -s12 /etc/ld.so.cache|head -5
00745a /usr/lib/libz.so.1
007475 /usr/lib/libz.so
007486 libxvidcore.so.4
007497 /usr/lib/libxvidcore.so.4
0074b1 libxslt.so.1
```

なお、-t オプションと -s オプションは同時に使うことはできません。

od の利用例

画像などのバイナリファイルをソースコードに含みたい場合など、バイナリファイルをダンプして適当なCの配列に変換したいことがあります。その時にodおよびsedだけで次のようにすることができます。

```
#!/bin/sh
# $0 objname < in > out
objname=${1:-objname}
od -A n -v -t x1 | sed -e '1i\
const unsigned char '$objname'[] = {
s/\([0-9a-f][0-9a-f]\) */0x\1,/g
$s/,$//
$a\
};
'
```

スクリプトの第一引数が配列名になります。標準入力からバイナリファイルを読みとって、C の配列を標準出力に出力します。

```
od -A n -v -t x1
```

を使うことで、標準入力の内容を下のようにしてダンプすることになります。

- オフセットは表示しない(-A n)
- 省略しないで全部ダンプする(-v)
- 1 バイトずつ 16 進数でダンプする(-t x1)

その出力を sed を使って次のように C の配列になるようにしています。

- 「const unsigned char 配列名[] = {」を最初の行に入れる
- 1 バイトずつ「0xNN,」という形式に変換する
- 最後の行の最後の「,」を削除する

- 最後に「};」という行を追加する

まとめ

本Hackではodを使うと、8進数ダンプだけでなくいろいろなフォーマットでダンプできることを説明しました。stringsのようにバイナリファイルに含まれている文字列もダンプすることができます。

—— Fumitoshi Ukai

2章

オブジェクトファイル Hack

Hack #5-21

通常、オブジェクトファイルはコンパイラが生成する中間ファイルを指しますが、本章ではオブジェクトファイルを広義に捉えて、実行ファイルやライブラリも含むことにします。オブジェクトファイルはマシン語のコードだけではなく、シンボルテーブル、デバッグ情報、再配置情報といったさまざまな情報を含んでいます。

本章では、オブジェクトファイルに含まれる情報を取り出すさまざまな方法や、オブジェクトファイルを書き換えて不要な情報を削除したり、データを埋め込んだりする方法を紹介します。オブジェクトファイルに親しむことはBinary Hackを身に付ける上での第一歩です。

HACK #5

ELF 入門

このHackでは、バイナリオブジェクトや実行ファイルのフォーマットであるELFについて説明します。

ELF(Executable and Linking Format)

ELF とは Executable and Linking Format の略で、実行可能バイナリやオブジェクトファイルなどのフォーマットを規定したものです。ELFフォーマットのファイルは、ELFヘッダが先頭にあり、プログラムヘッダテーブルおよびセクションヘッダテーブルがその後にあります。

これらのヘッダの構造は elf.h に記述されています。

ELF で使う型

ELFバイナリには32ビットと64ビットのものがあります。ELFでは次のような型を利用しています。N の部分は 32 ビットバイナリなら 32、64 ビットバイナリなら 64 となります。たとえば、32 ビットバイナリなら Elf32_Half、64 ビットバイナリなら Elf64_Half です。

型名	N=32	N=64	説明
ElfN_Half	uint16_t	uint16_t	符号なし16ビット値
ElfN_Word	uint32_t	uint32_t	符号なし32ビット値
ElfN_Sword	int32_t	int32_t	符号付き32ビット値
ElfN_Xword	uint64_t	uint64_t	符号なし64ビット値
ElfN_Sxword	int64_t	int64_t	符号付き64ビット値
ElfN_Addr	uint32_t	uint64_t	アドレス
ElfN_Off	uint32_t	uint64_t	オフセット
ElfN_Section	uint16_t	uint16_t	セクションインデックス
ElfN_Versym	uint16_t	uint16_t	バージョンシンボル情報

ELFヘッダ

ELFヘッダはELFファイルの先頭に必ず存在し、そのファイルがELFファイルであることを表します。ELFヘッダの内容はreadelfの-hオプション(--file-headerオプション)で見ることができます。

```
% readelf -h /bin/ls
ELF ヘッダ:
 マジック:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 クラス:                            ELF32
 データ:                            2 の補数、リトルエンディアン
 バージョン:                        1 (current)
 OS/ABI:                            UNIX - System V
 ABI バージョン:                    0
 タイプ:                            EXEC (実行可能ファイル)
 マシン:                            Intel 80386
 バージョン:                        0x1
 エントリポイントアドレス:          0x8049a50
 プログラムの開始ヘッダ:            52 (バイト)
 セクションヘッダ始点:              74948 (バイト)
 フラグ:                            0x0
 このヘッダのサイズ:                52 (バイト)
 プログラムヘッダサイズ:            32 (バイト)
 プログラムヘッダ数:                8
 セクションヘッダ:                  40 (バイト)
 Number of section headers:         25
 Section header string table index: 24
```

ELFヘッダは次のような構造をしています。

```
unsigned char e_ident[EI_NIDENT];  /* マジック
                                    * クラス,データ,バージョン
                                    * OS/ABI,ABIバージョン
                                    */
```

```
ElfN_Half   e_type;         /* タイプ */
ElfN_Half   e_machine;      /* マシン */
ElfN_Word   e_version;      /* バージョン */
ElfN_Addr   e_entry;        /* エントリポイントアドレス */
ElfN_Off    e_phoff;        /* プログラムヘッダ始点 */
ElfN_Off    e_shoff;        /* セクションヘッダ始点 */
ElfN_Word   e_flags;        /* フラグ */
ElfN_Half   e_ehsize;       /* このヘッダのサイズ */
ElfN_Half   e_phentsize;    /* プログラムヘッダサイズ */
ElfN_Half   e_phnum;        /* プログラムヘッダ数 */
ElfN_Half   e_shentsize;    /* セクションヘッダサイズ */
ElfN_Half   e_shnum;        /* セクションヘッダ数 */
ElfN_Half   e_shstrndx;     /* セクション名のストリングテーブル */
```

e_ident は ELF のマジックナンバーとその他の情報を保持しています。ELF ファイルは先頭 4 バイトが次のようなマジックナンバーを持っています。

```
| 0x7F | 0x45 | 0x4C | 0x46 |
```

これを文字列として表すと "\177ELF" となります。

その次のバイトで 32 ビットの場合は ELFCLASS32 (1)、64 ビットの場合は ELFCLASS64 (2)となります。その次のバイトはエンディアンを表しており、リトルエンディアンの場合は ELFDATA2LSB (1)、ビッグエンディアンの場合は ELFDATA2MSB (2)を使います。その後に ELF バージョンや OS、ABI などの情報を 1 バイトずつ使って表しています。

e_type は次のタイプのどれかを表しています。

型名	値	説明
ET_REL	1	リロケータブルファイル
ET_EXEC	2	実行可能ファイル
ET_DYN	3	共有オブジェクトファイル
ET_CORE	4	コアファイル

e_machine は、アーキテクチャタイプを表します。EM_ ではじまる定数として定義されています。

e_version は、ELF バージョンを表します。現在は EV_CURRENT (1)です。

e_entry は、この ELF で実行開始する仮想アドレスです。

e_ehsize は、ELF ヘッダ自体のサイズを表しています。

e_phoff、e_phentsize、e_phnumで、プログラムヘッダテーブルがどこにいくつあるかを表しています。

e_shoff、e_shentsize、e_shnumで、セクションヘッダテーブルがどこにいくつあるかを表しています。

e_shstrndxはセクション名のストリングテーブルを持つセクションヘッダインデックスを表しています。

プログラムヘッダ

プログラムヘッダテーブルはELFヘッダのe_phoffで指定されるオフセットからはじまり、e_phentsizeとe_phnumで決まる大きさのテーブルからなります。e_phentsizeがテーブルのなかのプログラムヘッダのサイズを表し、e_phnumがそのテーブルの中にいくつセクションヘッダがあるかを表しています。プログラムヘッダテーブル自体はe_phentsize * e_phnumバイト分あります。

プログラムヘッダはreadelfの-lオプション(--program-headers)で表示されます。

```
% readelf -l /bin/ls

Elf ファイルタイプは EXEC (実行可能ファイル) です
エントリポイント 0x8049a50
8 個のプログラムヘッダ、始点オフセット 52

Program Headers:
 タイプ           オフセット 仮想Addr    物理Addr    FileSiz MemSiz  Flg Align
 PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
 INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
     [要求されるプログラムインタプリタ: /lib/ld-linux.so.2]
 LOAD           0x000000 0x08048000 0x08048000 0x11d08 0x11d08 R E 0x1000
 LOAD           0x012000 0x0805a000 0x0805a000 0x003f4 0x007b0 RW  0x1000
 DYNAMIC        0x012184 0x0805a184 0x0805a184 0x000d8 0x000d8 RW  0x4
 NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
 GNU_EH_FRAME   0x011cdc 0x08059cdc 0x08059cdc 0x0002c 0x0002c R   0x4
 GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

セグメントマッピングへのセクション:
 セグメントセクション...
  00
  01     .interp
  02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr
  03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
  04     .dynamic
  05     .note.ABI-tag
  06     .eh_frame_hdr
  07
```

プログラムヘッダは次のような構造をしています。

```
ElfN_Word    p_type;          /* セグメントタイプ */
ElfN_Off     p_offset;        /* セグメントオフセット */
ElfN_Addr    p_vaddr;         /* 仮想Addr */
```

```
    ElfN_Addr    p_paddr;        /* 物理 Addr */
    ElfN_Word    p_filesz;       /* ファイルサイズ(FileSiz) */
    ElfN_Word    p_memsz;        /* メモリサイズ(MemSiz) */
    ElfN_Word    p_flags;        /* フラグ (Flg) */
    ElfN_Word    p_align;        /* アライメント(Align) */
```

これらは readelf -l で表示される Program Headers の個々の行に対応しています。
タイプ(p_type)には次のようなものがあります。

p_type	値	説明
PT_LOAD	1	ロードされるプログラムセグメント
PT_DYNAMIC	2	動的リンク情報
PT_INTERP	3	プログラムインタープリタ
PT_NOTE	4	補助的な情報
PT_PHDR	6	プログラムヘッダテーブル自体
PT_TLS	7	スレッドローカルストレージ
PT_GNU_EH_FRAME	0x6474e550	GNU .eh_frame_hdr セグメント
PT_GNU_STACK	0x6474e551	スタックの実行可能性

「セグメントマッピングへのセクション」以降の行で示されている情報は、Program Headers の個々のプログラムヘッダで示されるセグメントごとにそのセグメントのメモリ範囲に含まれるセクション名を並べています。つまり、最初(インデックス00)のプログラムヘッダで示されるセグメントはタイプがPHDRであり、それに含まれるセクションはありません。次(インデックス01)のプログラムヘッダで示されるセグメントはタイプがINTERPであり、それに含まれるセクションは.interpがあります。さらに次(インデックス02)のプログラムヘッダで示されるセグメントはタイプがLOADであり、この中には.interp、.note.ABI-tag、.hash、.dynsym、.dynstr、.gnu.version、.gnu.version_r、.rel.dyn、.rel.plt、.init、.plt、.text、.fini、.rodata、.eh_frame_hdrといったセクションが含まれています。

セクションヘッダ

セクションヘッダテーブルはELFヘッダのe_shoffで指定されるオフセットからはじまり、e_shentsizeとe_shnumで決まる大きさのテーブルからなります。e_shentsizeがテーブルのなかのセクションヘッダのサイズを表し、e_shnumがそのテーブルの中にいくつプログラムヘッダがあるかを表しています。プログラムヘッダテーブル自体はe_shentsize * e_shnumバイト分あります。

セクションヘッダはreadelfの-Sオプション(--section-headersオプション)で表示され

ます。

```
% readelf -S /bin/ls
25 個のセクションヘッダ、始点オフセット 0x124c4:

Section Headers:
[番] 名前              タイプ            アドレス  Off    サイズ  ES  Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0  0   0
[ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0  0   1
[ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0  0   4
[ 3] .hash             HASH            08048168 000168 000338 04   A  4  0   4
[ 4] .dynsym           DYNSYM          080484a0 0004a0 0006b0 10   A  5  1   4
[ 5] .dynstr           STRTAB          08048b50 000b50 00047b 00   A  0  0   1
[ 6] .gnu.version      VERSYM          08048fcc 000fcc 0000d6 02   A  4  0   2
[ 7] .gnu.version_r    VERNEED         080490a4 0010a4 0000b0 00   A  5  3   4
[ 8] .rel.dyn          REL             08049154 001154 000028 08   A  4  0   4
[ 9] .rel.plt          REL             0804917c 00117c 0002e0 08   A  4  b   4
[10] .init             PROGBITS        0804945c 00145c 000017 00  AX  0  0   4
[11] .plt              PROGBITS        08049474 001474 0005d0 04  AX  0  0   4
[12] .text             PROGBITS        08049a50 001a50 00c880 00  AX  0  0  16
[13] .fini             PROGBITS        080562d0 00e2d0 00001b 00  AX  0  0   4
[14] .rodata           PROGBITS        08056300 00e300 0039dc 00   A  0  0  32
[15] .eh_frame_hdr     PROGBITS        08059cdc 011cdc 00002c 00   A  0  0   4
[16] .data             PROGBITS        0805a000 012000 0000e8 00  WA  0  0  32
[17] .eh_frame         PROGBITS        0805a0e8 0120e8 00009c 00   A  0  0   4
[18] .dynamic          DYNAMIC         0805a184 012184 0000d8 08  WA  5  0   4
[19] .ctors            PROGBITS        0805a25c 01225c 000008 00  WA  0  0   4
[20] .dtors            PROGBITS        0805a264 012264 000008 00  WA  0  0   4
[21] .jcr              PROGBITS        0805a26c 01226c 000004 00  WA  0  0   4
[22] .got              PROGBITS        0805a270 012270 000184 04  WA  0  0   4
[23] .bss              NOBITS          0805a400 012400 0003b0 00  WA  0  0  32
[24] .shstrtab         STRTAB          00000000 012400 0000c3 00      0  0   1
Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)
```

セクションヘッダは次のような構造をしています。

```
ElfN_Word    sh_name;     /* 名前(ストリングテーブルでのインデックス) */
ElfN_Word    sh_type;     /* タイプ */
ElfN_Word    sh_flags;    /* フラグ(Flg)  */
ElfN_Addr    sh_addr;     /* アドレス */
ElfN_Off     sh_offset;   /* オフセット(Off) */
ElfN_Word    sh_size;     /* サイズ */
ElfN_Word    sh_link;     /* リンク(Lk) */
ElfN_Word    sh_info;     /* セクション情報(Inf) */
ElfN_Word    sh_addralign;/* アライメント(Al) */
ElfN_Word    sh_entsize;  /* セクションがテーブルの場合、個々のエントリのサイズ */
```

これらは readelf -S で表示される Section Headers の個々の行に対応しています。

名前は、ELFヘッダのe_shstrndxで指定されているセクションに含まれているストリングテーブルのインデックスで指定されています。この/bin/lsの例の場合、e_shstrndxは24だったので、24番目のセクションヘッダが、そのストリングテーブルを保持しているセクションとなります。

```
  [24] .shstrtab         STRTAB          00000000 012400 0000c3 00      0   0  1
```

これを見るとsh_offsetが0x012400で、サイズが0x0000c3バイトのストリングテーブルであることがわかります。

セクションタイプとしては以下のものがあります。

セクションタイプ	値	説明
SHT_PROGBITS	1	プログラムデータ
SHT_SYMTAB	2	シンボルテーブル
SHT_STRTAB	3	ストリングテーブル
SHT_RELA	4	加数付きのリロケーションエントリ
SHT_HASH	5	シンボルハッシュテーブル
SHT_DYNAMIC	6	動的リンク情報
SHT_NOTE	7	Notes
SHT_NOBITS	8	ファイル上にデータのない部分(.bss)
SHT_REL	9	リロケーションエントリ
SHT_DYNSYM	11	動的リンカが使うシンボルテーブル
SHT_INIT_ARRAY	14	コンストラクタの配列(.init)
SHT_FINI_ARRAY	15	デストラクタの配列(.fini)
SHT_GNU_verdef	0x6ffffffd	バージョン定義セクション
SHT_GNU_verneed	0x6ffffffe	バージョン要求セクション
SHT_GNU_versym	0x6fffffff	バージョンシンボルテーブル

ストリングテーブル

ストリングテーブルは単純な文字列のリストです。/bin/lsの場合、次のセクションがストリングテーブルです。

```
  [ 5] .dynstr           STRTAB          08048b50 000b50 00047b 00   A  0   0  1
  [24] .shstrtab         STRTAB          00000000 012400 0000c3 00      0   0  1
```

例として24の.shstrtabを見てみましょう。.shstrtabのオフセットは0x012400でサイズが

0xc3なのでodを使うと次のようにして得ることができます。

```
% od --skip-bytes 0x12400 --read-bytes 0xc3 -t x1z /bin/ls
0222000 00 2e 73 68 73 74 72 74 61 62 00 2e 69 6e 74 65  >..shstrtab..inte<
0222020 72 70 00 2e 6e 6f 74 65 2e 41 42 49 2d 74 61 67  >rp..note.ABI-tag<
0222040 00 2e 68 61 73 68 00 2e 64 79 6e 73 79 6d 00 2e  >..hash..dynsym..<
0222060 64 79 6e 73 74 72 00 2e 67 6e 75 2e 76 65 72 73  >dynstr..gnu.vers<
0222100 69 6f 6e 00 2e 67 6e 75 2e 76 65 72 73 69 6f 6e  >ion..gnu.version<
0222120 5f 72 00 2e 72 65 6c 2e 64 79 6e 00 2e 72 65 6c  >_r..rel.dyn..rel<
0222140 2e 70 6c 74 00 2e 69 6e 69 74 00 2e 74 65 78 74  >.plt..init..text<
0222160 00 2e 66 69 6e 69 00 2e 72 6f 64 61 74 61 00 2e  >..fini..rodata..<
0222200 65 68 5f 66 72 61 6d 65 5f 68 64 72 00 2e 64 61  >eh_frame_hdr..da<
0222220 74 61 00 2e 65 68 5f 66 72 61 6d 65 00 2e 64 79  >ta..eh_frame..dy<
0222240 6e 61 6d 69 63 00 2e 63 74 6f 72 73 00 2e 64 74  >namic..ctors..dt<
0222260 6f 72 73 00 2e 6a 63 72 00 2e 67 6f 74 00 2e 62  >ors..jcr..got..b<
0222300 73 73 00                                         >ss.<
0222303
```

この場合、ストリングテーブルは次のようになっています。

インデックス	文字列
1	.shstrtab
11	.interp
19	.note

つまり、.shstrtabの先頭からのオフセットが、ストリングテーブルでのインデックスとなります。

シンボルテーブル

シンボルテーブルはシンボルとその値などを対応させるためのテーブルです。/bin/lsの場合、stripされているので動的シンボルテーブルのみがあります。シンボルテーブルはreadelfだと-sオプション(--symsオプション)で見ることができます。

```
% readelf -s /bin/ls

シンボルテーブル '.dynsym' は 107 個のエントリから構成されています:
 番号:    値  サイズ   タイプ  Bind    Vis 索引名
    0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
    1: 08049484    60 FUNC    GLOBAL DEFAULT  UND readlink@GLIBC_2.0 (2)
    2: 08049494   283 FUNC    GLOBAL DEFAULT  UND getgrnam@GLIBC_2.0 (2)
    3: 080494a4    42 FUNC    GLOBAL DEFAULT  UND __fpending@GLIBC_2.2 (3)
    4: 080494b4    58 FUNC    GLOBAL DEFAULT  UND acl_entries@ACL_1.0 (4)
(略)
```

これを ELF ヘッダから読みとってみましょう。

まずセクションヘッダからシンボルテーブルとしては次の .dynsym があることがわかります。

```
[ 4] .dynsym           DYNSYM          080484a0 0004a0 0006b0 10   A  5   1  4
```

これをダンプしてみると次のようになります。

```
% od --skip-bytes 0x4a0 --read-bytes 0x6b0 -t x1z /bin/ls
0002240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  >................<
0002260 6c 01 00 00 84 94 04 08 3c 00 00 00 12 00 00 00  >l.......<.......<
0002300 7d 03 00 00 94 94 04 08 1b 01 00 00 12 00 00 00  >}................<
0002320 3b 01 00 00 a4 94 04 08 2a 00 00 00 12 00 00 00  >;.......*.......<
0002340 49 00 00 00 b4 94 04 08 3a 00 00 00 12 00 00 00  >I.......:.......<
0002360 9a 02 00 00 c4 94 04 08 53 00 00 00 12 00 00 00  >........S.......<
0002400 fd 00 00 00 d4 94 04 08 ef 00 00 00 12 00 00 00  >................<
(略)
```

シンボルテーブルは次のような構造のテーブルです。ELF バイナリの 32 ビットと 64 ビットでは st_value のアライメントの都合で順序が変わっています。

- 32 ビット(16 バイト)

```
uint32_t        st_name;
Elf32_Addr      st_value;
uint32_t        st_size;
unsigned char   st_info;
unsigned char   st_other;
uint16_t        st_shndx;
```

- 64 ビット(24 バイト)

```
uint32_t        st_name;
unsigned char   st_info;
unsigned char   st_other;
uint16_t        st_shndx;
Elf64_Addr      st_value;
uint64_t        st_size;
```

st_nameはストリングテーブルでのインデックスを表しています。st_valueはシンボルの値です。st_size はシンボルのサイズです。st_info の下位 4 ビットはシンボルタイプなどの情報で次のようなものがあります。

シンボルタイプ	値	説明
STT_OBJECT	1	シンボルはデータオブジェクト
STT_FUNC	2	シンボルは実行コード
STT_SECTION	3	シンボルはセクションに関連づけられている
STT_FILE	4	シンボルの名前はそのオブジェクトに関連付けられたソースコードのファイル名
STT_COMMON	5	シンボルはコモンデータ
STT_TLS	6	シンボルはスレッドローカルデータ

またst_info の上位4ビットはそのシンボルのバインディングをどうするかを表します。

シンボルバインディング	値	説明
STB_LOCAL	0	そのシンボルはローカル
STB_GLOBAL	1	そのシンボルはグローバル
STB_WEAK	2	そのシンボルは weak

st_shndx は関連するセクションを表しています。

セクション	値	説明
SHN_UNDEF	0	未定義
SHN_ABS	0xfff1	絶対値をもつシンボル
SHN_COMMON	0xfff2	コモン用シンボル

/bin/ls の例を見てみましょう。0 番目の symbol 情報は次のように空です。

```
0002240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  >................<
```

次の symbol 情報は次のように読みます。

```
0002260 6c 01 00 00 84 94 04 08 3c 00 00 00 12 00 00 00  >l.......<.......<
```

アドレス0002260から始まるこのバイト列は次のような意味を持っています。最初の6c 01 00 00 は st_name = 0x16c に対応します。00 00 00 01 6c ではなく、6c 01 00 00 となっているのは、対象としているアーキテクチャ(x86)がリトルエンディアンであるためです。

```
st_name = 0x16c
st_value = 0x08049484
st_size = 0x3c == 60
st_info = 0x12 == (STB_GLOBAL | STT_FUNC)
st_other = 0
st_shndx = 0 == SHN_UNDEF
```

ここではシンボル名へのオフセット値 st_name は 0x16c となっています。

.dynstr セクションを見ると、ストリングテーブルは先頭から0xb50バイト目から始まっていることがわかります。

```
[ 5] .dynstr           STRTAB          08048b50 000b50 00047b 00   A  0   0  1
```

そこで、0xb50 に 0x16c を足した値である 0xcbc からはじまる文字列を見てみると、次のように readlink という文字列が見つかります。

```
% od --skip-bytes 0xcbc --read-bytes 16 /bin/ls
0006274 72 65 61 64 6c 69 6e 6b 00 5f 5f 6f 76 65 72 66  >readlink.__overf<
0006314
```

このように、st_name = 0x16cに対応するシンボルは"readlink"であることがわかりました。これは readelf の出力の以下の部分に対応しています。

```
1: 08049484    60 FUNC    GLOBAL DEFAULT  UND readlink@GLIBC_2.0 (2)
```

readelfの場合、バージョン情報なども調べているので表示される文字列は少し凝ったものになっています。

再配置情報

SHT_RELA もしくは SHT_REL というタイプを持つセクションは、再配置情報を持っています。SHT_RELA は次のような Rela 構造のテーブルを持っています。

```
ElfN_Addr      r_offset;
uintN_t        r_info;
intN_t         r_addend;
```

SHT_REL の場合は、r_addend がない次のような Rel 構造のテーブルを持っています。

```
ElfN_Addr     r_offset;
uintN_t       r_info;
```

r_offsetはリロケーションを行うべき場所を示す、セクションの先頭からのオフセットです。r_infoはリロケーションのタイプやシンボルテーブルのインデックスなどの情報を含んでいます。Relaの場合、リロケーションする場合に常に加算する値をr_addendとして持っています。

まとめ

UNIX系のOSで一般的に実行ファイルやオブジェクトファイルに使われているELFフォーマットについて説明しました。readelfコマンドやobjdumpコマンドを使えば簡単に調べることはできますが、バイナリハックを行う場合にはELFフォーマットを自分で読みとったりできるようになったほうが、いろいろとハックすることができるでしょう。

—— Fumitoshi Ukai

HACK #6 静的ライブラリと共有ライブラリ

本Hackでは静的ライブラリと共有ライブラリの違い、それぞれの特徴について説明します。

静的ライブラリ

静的ライブラリ(static library)は、さまざまなプログラムで使うような関数を含むオブジェクトファイルを1つのファイルとして扱えるようにまとめたものです。

プログラムを作成する時に、ソースファイルを分割してある程度のかたまりごとに別々にオブジェクトファイルにコンパイルして、それを最後にリンクして1つの実行可能ファイルを作成します。この場合、いくつものプログラムで利用されそうなモジュールが複数のオブジェクトファイルになっていると、それらをひとかたまりとして扱うのが面倒になります。

そこで考えだされたのがアーカイブファイル(.a)です。これは複数のオブジェクトファイルを1つのファイルにまとめたものです。ar(1)コマンドを使うことで複数のオブジェクトファイルを1つのアーカイブファイルに含めることができます。OSによってはranlib(1)を使うことで、このアーカイブ内のオブジェクトが提供しているシンボル情報のハッシュを作成し、アーカイブからシンボルを提供しているオブジェクトファイルの検索を効率よく行うこともあります†。このようなアーカイブファイルを静的ライブラリと呼びます。

静的ライブラリは、通常次のようにして作成します。

```
% cc -c -o foo.o foo.c
% cc -c -o bar.o bar.c
% ar ruv libfoo.a foo.o bar.o
ar: libfoo.a を作成します
a - foo.o
a - bar.o
```

ライブラリの内容はarコマンドで見ることができます。

† GNU/Linuxではranlibコマンドの実体はarコマンドと同じであり、ranlibを実行することはar -sを実行することとまったく同じです。

```
% ar tv libfoo.a
rw-r--r-- 1000/1000    639 Mar  1 02:48 2006 foo.o
rw-r--r-- 1000/1000    639 Mar  1 02:48 2006 bar.o
```

静的ライブラリをリンクする場合は、リンカは他のオブジェクトファイルで未定義なシンボルを見つけて指定された静的ライブラリの中からそのシンボルの定義をしているオブジェクトファイルのコピーを取り出して実行可能ファイルの中に含めてリンクを行います。

libfoo.aがライブラリのディレクトリにインストールされている場合、次のようにしてリンクを行います。

```
% cc -o baz baz.o -lfoo
```

この場合、baz.oの中の未定義のシンボルに対して、libfoo.aに含まれているオブジェクトの中で定義しているオブジェクトがあれば、それらを取りだして実行ファイルbazのほうにコピーしてリンクします。

ここでのポイントは、ライブラリの中のオブジェクトファイル単位で処理が行われること、リンクする時に実行可能ファイル内にオブジェクトファイルのコピーが含まれていることです。

静的ライブラリをリンクして作られた実行バイナリを実行する場合は、静的ライブラリはなくてもかまいません。必要なコードは実行バイナリにコピーされて含まれているからです。

共有ライブラリ

共有ライブラリ(shared library)は、共有されるという点で静的ライブラリとは異なります。OSの仮想メモリ管理システムの進歩により、1つのファイルをmmap(2)などを使って、複数のプロセスでメモリを共有して参照できるようになってきました。これを有効活用するようにしたものが共有ライブラリです。共有ライブラリは、共有オブジェクトとも呼ばれます。静的ライブラリの場合は、複数のオブジェクトファイルのアーカイブでしたが、共有ライブラリの場合は、複数のオブジェクトファイルを1つの巨大なオブジェクトファイルにしてそれらを共有できるようにしたものです。

昔のOSのメモリ管理では、巨大なファイルにしておくとプログラム実行前にそれらをまずメモリにロードする必要があったために効率はよくありませんでした。

最近のOSではとりあえずメモリマップだけ設定しておくだけで、実際にそのメモリ内容が参照されるまでディスクアクセスを遅延することができるので、巨大なオブジェクトファイルになっていても特に問題にはならなくなっています。

共有ライブラリを作る時は、通常次のようにします。

```
% cc -fPIC -c -o foo.o foo.c
% cc -fPIC -c -o bar.o bar.c
% cc -shared -Wl,-soname,libfoo.so.0 -o libfoo.so foo.o bar.o
```

共有ライブラリを作る時は、-sharedオプションをつけて共有オブジェクトを作ることになります。また一般的に-Wl,-sonameオプションによってリンカに、その共有オブジェクトに指定したSONAMEを指定しておきます。後述しますが、このSONAMEによってどの共有オブジェクトを実行時にリンクするかが決定されます。

共有ライブラリのリンクは、静的ライブラリと同じ手順で行うことができます。

```
% cc -o baz baz.o -lfoo
```

ただし実際に行われている処理はだいぶ異なります。この場合、baz.oの中の未定義シンボルに対して共有オブジェクトで定義されていれば、その共有オブジェクトのSONAMEを実行ファイルのNEEDEDに設定をするだけで共有オブジェクトに含まれているコード自体はコピーしません。

静的ライブラリとは違い、*.soの中にどのようなオブジェクトファイルがあるかは基本的には残りません。

ここでのポイントは、共有ライブラリ単位で処理が行われるということと、リンクする時には必要としている共有ライブラリのSONAMEだけを実行可能ファイルにNEEDEDとして登録してあるということです。

共有ライブラリをリンクした実行ファイルを実行する時に、動的リンカローダ(ld.so)がNEEDEDの情報を使って必要としている共有ライブラリを探し出し、実行時にそのプロセスのメモリマップを操作して共有ライブラリと実行バイナリを同じプロセス空間で使えるようにしています。したがって、共有ライブラリをリンクした実行ファイルを実行する時には、共有ライブラリがシステムに存在している必要があります。実際のライブラリのコードは実行ファイルのほうには含まれておらず共有ライブラリのほうにしか存在しないからです。

ファイルサイズ

ファイルサイズという観点から見ると、共有ライブラリのほうがシステム全体としては小さくてすみます。静的ライブラリを使っている場合、ライブラリに含まれているコードをさまざまな実行ファイルで利用していると、それらがコピーされるために必要な容量が増えてしまうためです。共有ライブラリの場合は、ライブラリのコード自体はコピーされず共有ライブラリだけが持っていることから、ライブラリのコードを利用する実行ファイルがたくさんあっても、実行ファイルごとのコードの分が必要なだけで、ライブラリの分が実行ファイルごとに増えていくことはありません。

メモリサイズ

実行時に必要となるメモリサイズも、最近のOSでは共有ライブラリのほうが有利です。特にPICコード(Position Independent Code)にしておけば、コード部分はどのアドレスに配置しても変更する必要がないために、共有ライブラリを1つの物理メモリページに読み込むだけで、それぞれ別のメモリ空間にあるプロセスからその共有ライブラリのメモリページを共有することができるからです。

ライブラリへのパッチ

最近はセキュリティホールなどが、ライブラリに見つかることもたびたびあります。そのような場合にライブラリのセキュリティホールを修正したライブラリに入れかえる必要があります。

静的ライブラリを使っている場合は、その静的ライブラリを作りなおすだけでは十分ではありません。そのライブラリを使ってコンパイルされた実行バイナリのほうにコピーされたものが残っているので、そのライブラリを使っている実行バイナリすべてを再コンパイルする必要があります。

共有ライブラリを使っている場合は、その共有ライブラリにのみ問題のコードがあるので共有ライブラリを入れかえるだけで済みます†。もちろんデーモンのように長時間実行しているプログラムの場合は、以前の共有ライブラリがすでにメモリにマップされているので、新しい共有ライブラリを参照するように再起動する必要があるでしょう。

まとめ

本Hackでは静的ライブラリと共有ライブラリの違い、それぞれの特徴について説明しました。静的ライブラリは、単なるオブジェクトファイルのアーカイブであり、ライブラリをリンクする場合はそれらのオブジェクトファイルはコピーされています。コピーが実行ファイルに含まれるために実行時には静的ライブラリはなくてもかまいません。共有ライブラリは、オブジェクトファイルをまとめた巨大なオブジェクトファイルであり、ライブラリをリンクする時はSONAMEで参照する情報を実行ファイルに含めているだけです。実行ファイルには、どのライブラリを使うかという情報しか含まれていないため、実行時にはリンク時に使っていた共有ライブラリが存在している必要があります。

—— Fumitoshi Ukai

† 共有ライブラリのソースコードの変更が実行バイナリにも変更が及ぶ場合、例えばABIが変わったりマクロやインライン関数などに変更が加わっている場合にはその共有ライブラリを利用しているコードも再コンパイルする必要があります。

HACK #7 lddで共有ライブラリの依存関係をチェックする

本Hackでは共有ライブラリの依存関係を調べる方法を紹介します。

共有ライブラリの依存関係

共有ライブラリを利用する実行ファイル、および共有ライブラリそのものは、それらを実行しようとする時に必要となる別の共有ライブラリが何かといった情報を持っています。その情報はELFの「動的セクション」のNEEDEDに記録されています。例えば/bin/lsの場合、objdumpコマンドを使うと次のようにして見ることができます。

```
% objdump -p /bin/ls
/bin/ls:     ファイル形式 elf32-i386

プログラムヘッダ:
(略)

動的セクション:
  NEEDED      librt.so.1
  NEEDED      libacl.so.1
  NEEDED      libc.so.6
  INIT        0x804945c
(略)
```

readelfコマンドの場合は次のようにして見ることができます。

```
% readelf -d /bin/ls

Dynamic segment at offset 0x12184 contains 22 entries:
 タグ        タイプ                         名前/値
 0x00000001 (NEEDED)                       共有ライブラリ: [librt.so.1]
 0x00000001 (NEEDED)                       共有ライブラリ: [libacl.so.1]
 0x00000001 (NEEDED)                       共有ライブラリ: [libc.so.6]
 0x0000000c (INIT)                         0x804945c
(略)
```

このように/bin/lsは、librt.so.1、libacl.so.1、libc.so.6の3つの共有ライブラリを必要としていることがわかります。

しかし、/bin/lsを実行する場合に必要なのは、この3つの共有ライブラリだけではありません。この3つの共有ライブラリ自体がそれぞれ必要としている別の共有ライブラリも必要となります。

NEEDEDで記述されているのはSONAMEなので、SONAMEから実際のファイルを探してくる必要があります。特に設定されていない場合は/usr/libおよび/libからそのSONAMEに対応するファイルが存在する場合、それがその共有ライブラリです[†]。環境変数LD_LIBRARY_PATHにラ

イブラリのパスを設定している場合、そのディレクトリを参照します。また/etc/ld.so.cacheに記録されている情報にあれば、そちらを参照します。/etc/ld.so.cacheは、/etc/ld.so.confの設定を使ってldconfigを実行する時に更新されます。

例えば、librt.so.1の場合、/lib/librt.so.1というファイル(シンボリックリンク)が存在しているのでそれがSONAME librt.so.1に対応する共有ライブラリのファイルです。この共有ライブラリ自体の依存ライブラリも同様にobjdumpやreadelfで見ることができます。

```
% readelf -d /lib/librt.so.1

Dynamic segment at offset 0x61b8 contains 25 entries:
  タグ        タイプ                         名前/値
 0x00000001 (NEEDED)                     共有ライブラリ: [libc.so.6]
 0x00000001 (NEEDED)                     共有ライブラリ: [libpthread.so.0]
 0x00000001 (NEEDED)                     共有ライブラリ: [ld-linux.so.2]
 0x0000000e (SONAME)                     ライブラリの soname: [librt.so.1]
 0x0000000c (INIT)                       0x177c
(略)
```

このようにlibrt.so.1は、libc.so.6、libpthread.so.0、ld-linux.so.2を必要としていることがわかります。

同様に他の共有ライブラリも調べてみると次のようになります。

```
/bin/ls  NEEDED  librt.so.1   NEEDED libc.so.6
                              NEEDED libpthread.so.0 NEEDED libc.so.6
                                                     NEEDED ld-linux.so.2
                              NEEDED ld-linux.so.2
         NEEDED  libacl.so.1  NEEDED libattr.so.1    NEEDED libc.so.6
                              NEEDED libc.so.6
         NEEDED  libc.so.6    NEEDED ld-linux.so.2
```

よって、/bin/lsを実行するために必要となる共有ライブラリは、librt.so.1、libacl.so.1、libc.so.6、libpthread.so.0、libattr.so.1、ld-linux.so.2ということがわかります。

lddを使って共有ライブラリの依存関係をチェックする

以上のように、objdumpやreadelfを使って共有ライブラリの依存関係をチェックすることは不可能ではありませんが、個々の共有ライブラリに関してすべての依存関係を調べていく必要があるので面倒です。また、実際に実行される時にどのディレクトリにある共有ライブラリが使われるかという点で正確な結果が得られているとは限りません。

lddコマンドを使えば以上の処理をまとめて実行してくれます。

† 実際には/lib/tlsや/lib/tls/i686/cmovなどにある共有ライブラリが使われることがある。

```
% ldd /bin/ls
   librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7fd2000)
   libacl.so.1 => /lib/libacl.so.1 (0xb7fcb000)
   libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e96000)
   libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e86000)
   /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
   libattr.so.1 => /lib/libattr.so.1 (0xb7e82000)
```

このように実行ファイルが必要とする共有ライブラリのSONAME、およびそのパス名とそれが割り当てられるメモリアドレスが一覧表示されます。

lddは実行ファイル以外にも共有ライブラリに対して使うことができます。

```
% ldd /lib/librt.so.1
   libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ea7000)
   libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e97000)
   /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

GNU/Linuxでは、lddは実は単なるシェルスクリプトです。ポイントは環境変数LD_TRACE_LOADED_OBJECTSです。環境変数LD_TRACE_LOADED_OBJECTSに1を設定してプログラムを実行すると、プログラムを実行開始時にELFインタプリタ(ランタイムローダ/lib/ld-linux.so.2)が、必要な共有ライブラリを調べてメモリにマップしてその情報を表示して実際のプログラムを実行する前に終了しているのです。したがってlddを使わなくても環境変数LD_TRACE_LOADED_OBJECTSを使えば同じような結果を得ることができます。

```
% LD_TRACE_LOADED_OBJECTS=1 /bin/ls
  librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7fd2000)
  libacl.so.1 => /lib/libacl.so.1 (0xb7fcb000)
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e96000)
  libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e86000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
  libattr.so.1 => /lib/libattr.so.1 (0xb7e82000)
```

実行ファイルではなくて、共有ライブラリの場合は実行できないので同じようにすることはできません。

```
% LD_TRACE_LOADED_OBJECTS=1 /lib/librt.so.1
zsh: 許可がありません: /lib/librt.so.1
```

この場合は、ランタイムローダ/lib/ld-linux.so.2を実行します。

```
% LD_TRACE_LOADED_OBJECTS=1 /lib/ld-linux.so.2 /lib/librt.so.1
  libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7ea7000)
  libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e97000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

まとめ

共有ライブラリの依存関係は、必要となる共有ライブラリのSONAMEが、実行ファイルおよび共有ライブラリのELFの動的セクションのNEEDEDに記録されることで管理されています。個々のファイルのNEEDEDはobjdumpやreadelfを使って見ることができますが、依存関係を満たすすべての共有ライブラリを表示するにはlddコマンドを使います。lddコマンドは、環境変数LD_TRACE_LOADED_OBJECTS を使ってこれを実現しています。

—— Fumitoshi Ukai

HACK #8 readelf で ELF ファイルの情報を表示する

本 Hack では ELF ファイルの情報を表示するツール readelf について説明します。

readelfはBFDライブラリを使わずに直接ELFを読むためのツールです。BFDに依存しないプログラムがあることで、ELFファイルの問題かBFDの問題かの切り分けがしやすくなります。readelf は、BFD を経由しないで ELF ファイルを読んでいるので objdump よりも詳細な情報を得ることができます。例えば DWARF デバッグ情報などを調べることができます。

readelfはどの情報を読み出すかを、必ず何らかのオプションで指示しないといけません。オプションを指定しない場合は利用法が表示されます。

ELF ヘッダの読み出し

ELF ヘッダを読み出すオプションは次の通りです。

見たいヘッダ	オプション	ロングオプション
ELF ファイルヘッダ	-h	--file-header
プログラムヘッダ	-l	--program-headers、--segments
セクションヘッダ	-S	--section-headers、--sections
以上の 3 つのヘッダ	-e	--headers

ELF 情報の読み出し

見たい情報	オプション	ロングオプション
シンボルテーブル	-s	--syms、-symbols
リロケーション情報	-r	--relocs
ダイナミックセグメント	-d	--dynamic
バージョンセクション	-V	--version-info

見たい情報	オプション	ロングオプション
アーキテクチャ依存	-A	--arch-specific
バケットリスト長のヒストグラム	-I	--histogram
ヘッダすべてと以上のすべて	-a	--all
コアノート(core notes)	-n	--notes
unwind 情報	-u	--unwind

通常、シンボル情報はシンボルセクションにあるシンボル情報を使いますが、-Dオプション(--use-dynamicオプション)を使うと、ダイナミックセクションにあるシンボル情報としてシンボル情報を使うようになります。

ELF セクションのダンプ

-x オプション(--hex-dump オプション)で指定したセクションの内容をダンプします。

セクションはセクション番号で指示します。セクション番号は-Sオプションで表示されるセクションヘッダに付いている番号です。

```
% readelf -S /bin/ls
25 個のセクションヘッダ、始点オフセット 0x124c4:

Section Headers:
 [番] 名前              タイプ      アドレス   Off    サイズ   ES  Flg  Lk  Inf Al
 [ 0]                   NULL        00000000 000000 000000 00      0   0   0
 [ 1] .interp           PROGBITS    08048134 000134 000013 00   A  0   0   1
 [ 2] .note.ABI-tag     NOTE        08048148 000148 000020 00   A  0   0   4
 [ 3] .hash             HASH        08048168 000168 000338 04   A  4   0   4
 [ 4] .dynsym           DYNSYM      080484a0 0004a0 0006b0 10   A  5   1   4
 [ 5] .dynstr           STRTAB      08048b50 000b50 00047b 00   A  0   0   1
(略)
```

.interp はセクション番号が1なので、その内容を見るためには、次のようにします。

```
% readelf -x1 /bin/ls

セクション '.interp' の 16 進数ダンプ:
 0x08048134 6f732e78 756e696c 2d646c2f 62696c2f /lib/ld-linux.so
 0x08048144                                    00322e .2.
```

.note.ABI-tag ならセクション番号は2、.hash ならセクション番号は3となるわけです。

DWARF2 デバッグセクションの読み出し

-wオプション(--debug-dumpオプション)でDWARF2デバッグセクションの情報を表示します(DWARF2 については **[Hack #40]** を参照してください)。

-w	--debug-dump=	セクション
l	line	.debug_line
i	info	.debug_info
a	abbrev	.debug_abbrev
p	pubnames	.debug_pubnames
r	aranges	.debug_aranges
R	Ranges	.debug_ranges
m	macro	.debug_macinfo
f	frames	.debug_frame
F	frames-interp	.debug_frame
s	str	.debug_str
o	loc	.debug_loc

長いシンボルもすべて表示する

デフォルトでは、長いシンボルは表示が1行以内におさまるように後ろが切りとられてしまいます。-Wオプション(--wideオプション)を使うと80文字以上の長い出力もするようになります。

まとめ

readelf を使えば、ELF や DWARF の情報を解釈して表示することができます。

—— Fumitoshi Ukai

HACK #9 objdumpでオブジェクトファイルをダンプする

本 Hack では、オブジェクトファイルをダンプするツールとしての objdump の使い方を説明します。

objdump で ELF バイナリをダンプする

objdumpは、どの情報を表示するかといったオプションを指定する必要があります。単にダンプする場合は、-s オプション(--full-contents オプション)を使います。

```
% objdump -s /bin/ls
/bin/ls:     ファイル形式 elf32-i386
セクション .interp の内容:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
セクション .note.ABI-tag の内容:
 8048148 04000000 10000000 01000000 474e5500  ............GNU.
 8048158 00000000 02000000 02000000 00000000  ................
セクション .hash の内容:
 8048168 61000000 6b000000 00000000 3d000000  a...k.......=...
 8048178 3c000000 31000000 00000000 00000000  <...1...........
(略)
```

このように指定したELFバイナリのファイル形式および各セクションごとにその内容をダンプして表示してくれます。各セクションごとのダンプは、下のような形式で出力されています。

メモリアドレス 16進ダンプ(4バイト * 4) ASCII表示

ここで16進ダンプは、x86のようなリトルエンディアンアーキテクチャ上で実行しても、いわゆるビッグエンディアンで出力されています。例えば上記の場合、.interpの最初の4バイトはASCIIで"/lib"ですが、それの16進は2f 6c 69 62の順になっておりリトルエンディアンではありません。objdumpには--endianオプションがありますが、このオプションは単にobjdumpでディスアセンブルする場合に影響し、この出力には影響しません。

このように-sオプションしか指定していない場合はデフォルトのターゲットフォーマットでダンプした結果になります。通常はelf32-i386などになるので、セクションごとに識別してダンプするようになります。このターゲットフォーマットは-bオプション(--targetオプション)を使って変更できます。利用可能なターゲットフォーマットは-iオプション(--infoオプション)で調べられます。

```
% objdump -i
BFD ヘッダファイルバージョン 2.15
elf32-i386
 (header little endian, data little endian)
  i386
a.out-i386-linux
 (header little endian, data little endian)
  i386
(略)
```

objdumpでELFバイナリの特定セクションだけダンプする

特定セクションだけをダンプしたい場合は-jオプション(--sectionオプション)でセクション名を指定します。

```
% objdump -s -j .interp /bin/ls
/bin/ls:     ファイル形式 elf32-i386
セクション .interp の内容:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
%
```

どのようなセクションがあるかを調べるために、-sオプションだけで表示して「セクション ○○の内容:」の行を見るのもかまいませんが、-hオプション(--section-headersオプション、--headersオプション)で表示することもできます。

```
% objdump -h /bin/ls
/bin/ls:     ファイル形式 elf32-i386
セクション:
索引名               サイズ      VMA       LMA       File off  Algn
   0 .interp       00000013  08048134  08048134  00000134  2**0
           CONTENTS, ALLOC, LOAD, READONLY, DATA
   1 .note.ABI-tag 00000020  08048148  08048148  00000148  2**2
           CONTENTS, ALLOC, LOAD, READONLY, DATA
   2 .hash         00000338  08048168  08048168  00000168  2**2
           CONTENTS, ALLOC, LOAD, READONLY, DATA
 (略 )
```

ここの索引名のところ(.interpや.note.ABI-tagなど)がセクション名として使える名前です。

objdump でアドレス範囲を指定してダンプする

--start-addressオプション、および--stop-addressオプションを使えば、ダンプするアドレス範囲を指定することができます。例えば上の出力から.interpセクションはアドレス0x08048134から0x08048147までということがわかりますから、下のようにすることもできます。

```
% objdump -s --start-address=0x08048134 --stop-address=0x08048147 /bin/ls
/bin/ls:     ファイル形式 elf32-i386
セクション .interp の内容:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
セクション .note.ABI-tag の内容:
セクション .hash の内容:
セクション .dynsym の内容:
セクション .dynstr の内容:
セクション .gnu.version の内容:
セクション .gnu.version_r の内容:
セクション .rel.dyn の内容:
セクション .rel.plt の内容:
セクション .init の内容:
```

```
セクション .plt の内容:
セクション .text の内容:
セクション .fini の内容:
セクション .rodata の内容:
セクション .eh_frame_hdr の内容:
セクション .data の内容:
セクション .eh_frame の内容:
セクション .dynamic の内容:
セクション .ctors の内容:
セクション .dtors の内容:
セクション .jcr の内容:
セクション .got の内容:
%
```

セクションをまたいでアドレスを指定した場合などは、このようになります。

```
% objdump -s --start-address=0x08048134 --stop-address=0x08048150 /bin/ls
/bin/ls:     ファイル形式 elf32-i386
セクション .interp の内容:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
セクション .note.ABI-tag の内容:
 8048148 04000000 10000000                    ........
セクション .hash の内容:
セクション .dynsym の内容:
セクション .dynstr の内容:
セクション .gnu.version の内容:
セクション .gnu.version_r の内容:
セクション .rel.dyn の内容:
セクション .rel.plt の内容:
セクション .init の内容:
セクション .plt の内容:
セクション .text の内容:
セクション .fini の内容:
セクション .rodata の内容:
セクション .eh_frame_hdr の内容:
セクション .data の内容:
セクション .eh_frame の内容:
セクション .dynamic の内容:
セクション .ctors の内容:
セクション .dtors の内容:
セクション .jcr の内容:
セクション .got の内容:
%
```

objdumpで単純なバイナリをダンプする

そもそもELFではないファイルや、ELFファイルをELFではなく単なるバイナリファイルとしてダンプしてみたい場合には、ターゲットフォーマットとしてbinaryを指定します。

```
% objdump -s -b binary /bin/ls
/bin/ls:     ファイル形式 binary
セクション .data の内容:
 00000 7f454c46 01010100 00000000 00000000  .ELF............
 00010 02000300 01000000 509a0408 34000000  ........P...4...
 00020 c4240100 00000000 34002000 08002800  .$......4. ...(.
 00030 19001800 06000000 34000000 34800408  ........4...4...
 00040 34800408 00010000 00010000 05000000  4...............
 00050 04000000 03000000 34010000 34810408  ........4...4...
 00060 34810408 13000000 13000000 04000000  4...............
 00070 01000000 01000000 00000000 00800408  ................
 00080 00800408 081d0100 081d0100 05000000  ................
(略)
```

このようにファイル形式がbinaryとなり、セクションごとに分割して出力されないようになります。

ELF でないファイルの場合は次のようになります。

```
% objdump -s -b binary /etc/ld.so.cache
/etc/ld.so.cache:     ファイル形式 binary
セクション .data の内容:
 0000 6c642e73 6f2d312e 372e3000 18030000  ld.so-1.7.0.....
 0010 03000000 204f0000 2a4f0000 03000000  .... O..*O......
 0020 3d4f0000 454f0000 03000000 564f0000  =O..EO......VO..
 0030 674f0000 03000000 814f0000 8e4f0000  gO.......O...O..
 0040 03000000 a44f0000 b24f0000 03000000  .....O...O......
 0050 c94f0000 d54f0000 03000000 ea4f0000  .O...O.......O..
 0060 f84f0000 03000000 0f500000 22500000  .O.......P.."P..
 0070 03000000 3e500000 4a500000 03000000  ....>P..JP......
 0080 5f500000 6c500000 03000000 82500000  _P..lP.......P..
(略)
```

binary フォーマットは自動認識されないので必ずオプションに指定する必要があります。

```
% objdump -s /etc/ld.so.cache
objdump: /etc/ld.so.cache: ファイル形式が認識できません
```

この場合、アドレスはファイルオフセットと等しくなります。

まとめ

本Hackでは、objdumpをつかってELFバイナリ、または普通のファイルをダンプする方法を説明しました。

—— Fumitoshi Ukai

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 #11 objcopyで実行ファイルにデータを埋め込む

本Hackでは、objcopyを用いて実行ファイルにデータを埋め込む方法を紹介します。

プログラムの実行に不可欠なデータをファイルから読み込んで利用することがあります。この方法を用いると、データの更新が手軽にできるという利点がある一方で、単体の実行ファイルで実行できない、データファイルが紛失してしまう、といった問題もあります。本Hackではobjcopyを用いて実行ファイルにデータを埋め込む方法を紹介します。

データの埋め込み

小さなデータをソースコードに埋め込むのは簡単です。ソースコード中に埋め込まれている"hello, world"などのメッセージはソースコードに埋め込まれたデータと言えます。

一方、画像や辞書などの巨大なデータをソースコードに埋め込むのはそう簡単ではありません。まず、データを文字列などに変換する必要がある上に、変換後の巨大なソースコードはコンパイラが処理できるサイズを超えてしまう可能性があります。

objcopy

そこで登場するのがGNU binutilsに付属するobjcopyコマンドです。objcopyを使うと任意のファイルをリンク可能なオブジェクトファイルに変換できます。

例えば、foo.jpgをx86用のELF32形式のオブジェクトファイルfoo.oに変換するには次のように実行します。

```
% objcopy -I binary -O elf32-i386 -B i386 foo.jpg foo.o
```

foo.oをリンクしたCのプログラムからはfoo.jpgのデータは以下の変数名を用いて参照できます。

```
extern char _binary_foo_jpg_start[];
extern char _binary_foo_jpg_end[];
extern char _binary_foo_jpg_size[];
```

ポインタではなく配列なところがポイントです。これらの変数はたとえば次のように使います。

```
const char *start = _binary_foo_jpg_start;  // データの先頭のアドレスを取得
const char *end = _binary_foo_jpg_end;  // データの末尾のアドレス +1 を取得
int size = (int)_binary_foo_jpg_size;  // データのサイズを取得
```

最後の_binary_foo_jpg_sizeは、&_binary_foo_jpg_size[0]がアドレスではなく値(データのサイズ)となっているので要注意です。

まとめ

objcopyを用いて実行ファイルにデータを埋め込む方法を紹介しました。本Hackの例のような普通のデータだけでなく、自分自身のソースコードや別のプログラムのバイナリといった変なものを埋め込んで遊ぶのも面白いと思います。

—— Satoru Takabayashi

HACK #12 nmでオブジェクトファイルに含まれるシンボルをチェックする

本Hackではオブジェクトファイルに含まれるシンボルを見るツールnmの使い方を説明します。

nmの使い方

nmをオブジェクトファイルに対して実行すると、そのオブジェクトファイルに含まれているシンボルがリストアップされます。

```
% nm cabin.o
00003530 t .L1207
00003557 t .L1208
0000356c t .L1209
00003581 t .L1210
00003596 t .L1211
000035ab t .L1212
000044b9 t .L1427
00004561 t .L1428
```

nmはオブジェクトファイルに含まれているシンボルをアルファベット順に1行ずつ出力します。デフォルトでは出力フォーマットはbsdとなっていて、bsdフォーマットの場合は各行は、シンボルの値、シンボルクラス、シンボル名が出力されます。シンボルの値が決まっていないシンボル(シンボルクラスがU)に関してはシンボルの値は出力されずに空のカラムになります。

出力順序を逆順にするには-rオプション(--reverse-sortオプション)を使います。

```
% nm -r cabin.o
        U write
        U times
        U time
        U sysconf
```

```
         U strstr
         U strrchr
         U strlen
```

--size-sortオプションを使うとシンボルのサイズ(そのシンボルの指すオブジェクトのサイズ)を小さい順にソートするようになります。-rオプションを同時に使えばサイズの大きい順にソートすることになります。

```
% nm --size-sort cabin.o
00000001 T cbstdiobin
00000004 d asiz.7703
00000004 B cbfatalfunc
00000004 b farray.7701
00000004 b onum.7702
00000004 b parray.7699
00000007 T cbdatumptr
...
% nm --size-sort -r cabin.o
00000b5b T cbstrmktime
00000637 T cbmimebreak
000004ec T cburlbreak
0000036a T cbhsort
00000365 T cbxmlbreak
0000033f T cbdatestrhttp
0000033a T cbsprintf
000002b2 T cbmapputcat
0000026d T cbxmlattrs
...
```

ただし、未定義シンボルは出力されません。また、デフォルトの出力フォーマットでは--size-sortオプションを指定すると最初のカラムはシンボルのサイズが出力されるようになってしまいます。サイズでソートしてシンボルの値も表示する場合は、-Sオプションも同時に指定します。この場合、各行はシンボルの値、シンボルのサイズ、シンボルクラス、シンボル名の順になります。このようにしてオブジェクトファイルに含まれる巨大な関数やデータがどれかを調べることができます。

```
% nm --size-sort -r -S cabin.o
00007070 00000b5b T cbstrmktime
000053e0 00000637 T cbmimebreak
00005a20 000004ec T cburlbreak
00003970 0000036a T cbhsort
000035d0 00000365 T cbxmlbreak
00004350 0000033f T cbdatestrhttp
00006d30 0000033a T cbsprintf
00004c80 000002b2 T cbmapputcat
00005170 0000026d T cbxmlattrs
```

出力フォーマットは-fオプション(--formatオプション)で指定できます。デフォルトのbsdのほかにsysvやposixがあります。

```
% nm -f sysv foo.o
foo.o からのシンボル:

Name              Value    Class    Type         Size      Line   Section
change           |0000a130|   T   |           FUNC|0000009d|       |.text
check_buffer     |000081a0|   t   |           FUNC|00000045|       |.text
check_type       |        |   U   |         NOTYPE|        |       |*UND*
check_target     |00000124|   d   |         OBJECT|00000004|       |.data
clear_buffer     |000000f0|   D   |         OBJECT|00000004|       |.data

% nm -f posix foo.o
change T 0000a130 0000009d
check_buffer t 000081a0 00000045
check_type U
check_target d 00000124 00000004
clear_buffer D 000000f0 00000004
```

オブジェクトファイルを複数指定した場合は、各オブジェクトファイルごとにそのオブジェクトファイル内のシンボルをソートして出力します。-Aオプション(もしくは-oオプション、--print-file-nameオプション)を指定すると、各行の先頭にそのシンボルが、どのオブジェクトファイルにあったかファイル名を含めて出力するようになるので、grepなどでシンボルを探す時に、どのオブジェクトファイルかが簡単に調べられるようになります。

```
% nm -A *.o | grep check_target
foo.o:00000124 d check_target
```

実行ファイルについてもnmを使って、シンボルをチェックすることができます。

```
% nm a.out
080494e4 D _DYNAMIC
080495c0 D _GLOBAL_OFFSET_TABLE_
080484c0 R _IO_stdin_used
         w _Jv_RegisterClasses
080495b0 d __CTOR_END__
....
```

なお、nmで引数を省略した場合は、カレントディレクトリのa.outが対象になります。a.outはccが生成する実行バイナリのデフォルトのファイル名です。a.outがそのディレクトリにない場合は、エラーで終了します。

strip(1)を使って、実行ファイルのシンボルを切り捨ててから、nmで見ると次のようになっています。

```
% strip foo
% nm foo
nm: foo: シンボルがありません
```

ただし、静的バイナリでない場合は、共有ライブラリを動的にリンクするためのシンボル情報は残っています。このようなシンボルは動的シンボルなので-Dオプション(--dynamicオプション)を使う必要があります。

```
% nm -D foo
080484c0 R _IO_stdin_used
         w _Jv_RegisterClasses
         w __gmon_start__
         U __libc_start_main
         U printf
```

静的ライブラリについては、ライブラリファイルに含まれているオブジェクトファイルごとのシンボルを出力します。

```
% nm /usr/lib/libc.a

init-first.o:
         U __environ
         U __fpu_control
         U __init_misc
00000004 C __libc_argc
00000004 C __libc_argv
00000090 T __libc_init_first
         U __libc_init_secure
00000000 D __libc_multiple_libcs
         U __setfpucw
         U _dl_non_dynamic_init
000000a0 T _dl_start
         w _dl_starting_up
         U abort
00000000 t init

libc-start.o:
     U __close
     U __cxa_atexit
     U __environ
```

この場合も、-Aオプションを使うと、各行ごとにどのライブラリのどのオブジェクトファイルかを出力するようになります。

```
% nm -A /usr/lib/libc.a
/usr/lib/libc.a:init-first.o:         U __environ
/usr/lib/libc.a:init-first.o:         U __fpu_control
/usr/lib/libc.a:init-first.o:         U __init_misc
```

```
/usr/lib/libc.a:init-first.o:00000004 C __libc_argc
/usr/lib/libc.a:init-first.o:00000004 C __libc_argv
/usr/lib/libc.a:init-first.o:00000090 T __libc_init_first
```

共有ライブラリの場合は、1つの巨大なオブジェクトファイルのように見えます。

```
% nm libqdbm.so.11.5.0
00005588 t .L10
0000d148 t .L10
0000558f t .L11
0000d14f t .L11
00005596 t .L12
0000d156 t .L12
```

共有ライブラリも、実行バイナリと同様、インストール時はstripされてしまっていることが多いので、システムにインストールされている共有ライブラリの場合は次のように出力されるでしょう。

```
% nm /usr/lib/libqdbm.so.11.5.0
nm: /usr/lib/libqdbm.so.11.5.0: シンボルがありません
```

この場合も-Dオプションを使うことで動的シンボルを見ることができます。

```
% nm -D /usr/lib/libqdbm.so.11.5.0
00027028 D VL_CMPDEC
00027020 D VL_CMPINT
0002701c D VL_CMPLEX
00027024 D VL_CMPNUM
0002703c D VST_CMPDEC
00027034 D VST_CMPINT
00027030 D VST_CMPLEX
00027038 D VST_CMPNUM
00027068 A _DYNAMIC
00027154 A _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
```

シンボルクラス

nmの出力を読みとくには、シンボルクラスの意味を理解しなければなりません。シンボルにはいくつかのクラスがあり、nmではそれらを1文字で表現しています。シンボルクラスには以下のようなものがあります。大文字、小文字の区別は意味があり、大文字はグローバルな(外部参照されうる)シンボルで、小文字はファイルローカルなシンボルです。

アルファベット順に見ていくと次の表のようになります。

シンボルクラス	説明
A	シンボルの値が絶対値、つまりリンクしても変化しない
B	シンボルは未初期化データ領域(BSS)にある
C	共有(common)のシンボル。未初期化データ
D	シンボルは初期化済みデータセクションにある
G	シンボルは小さなオブジェクトで使われる初期化済みデータセクションにある(近くのシンボルはより効率よくアクセスできる場合があるため)
I	シンボルは別のシンボルへの間接参照。a.out の GNU 拡張
N	デバッグ用シンボル
R	シンボルは読み込み専用データセクションにある
S	シンボルは小さなオブジェクトにつかわれる未初期化データセクションにある
T	シンボルはテキスト(コード)セクションにある
U	未定義シンボル。別のオブジェクトファイルもしくは共有ライブラリにシンボルの実体があるはず
V	シンボルはウィーク(weak)オブジェクト
W	シンボルは weak オブジェクトシンボルと決まっていない weak シンボル
-	シンボルは a.out オブジェクトファイル内の stabs シンボル(デバッグ情報など)
?	未知のシンボルクラス

セクションごとに分類すると次のようになります。

セクション	シンボルクラス	ローカルかグローバルか(参照可能範囲)
テキストセクション	T	グローバル
	t	ローカル
データセクション	D	グローバル
	G	グローバル(小さなオブジェクト用)
	d	ローカル
	g	ローカル(小さなオブジェクト用)
読み込み専用データ	R	グローバル
	r	ローカル
BSS(未初期化データ)	B	グローバル
	S	グローバル(小さなオブジェクト用)
	b	ローカル
	s	ローカル(小さなオブジェクト用)
weak オブジェクト	V	グローバル
	v	ローカル
weak シンボル	W	グローバル

セクション	シンボルクラス	ローカルかグローバルか(参照可能範囲)
	w	ローカル
コモン	C	グローバル
デバッグ用	N	グローバル
	n	ローカル
	-	stabs
絶対値	A	グローバル
	a	ローカル
未定義	U	グローバル
間接参照	I	グローバル
	i	ローカル
未知のクラス	?	

まとめ

本Hackでは、オブジェクトファイルに含まれるシンボルを調べる基本となるツールnmについて説明しました。

—— Fumitoshi Ukai

HACK #13 stringsでバイナリファイルから文字列を抽出する

本Hackではstringsの使い方と仕組みについて紹介します。

stringsはバイナリファイルから文字列を抽出するためのツールです。GNU Binutilsに含まれています。本Hackではstringsの使い方と仕組みを紹介します。

strings の使い方

stringsの基本的な使い方は簡単です。文字列を抽出したいバイナリファイルを引数に渡せばOKです。バイナリファイルは/bin/lsのような実行ファイルでもfoo.jpg、bar.mp3のような任意のバイナリファイルでもOKです。

```
% strings /bin/ls | head -5
/lib/ld-linux.so.2
librt.so.1
clock_gettime
_Jv_RegisterClasses
__gmon_start__
```

標準入力からバイナリデータを読み込んで処理することもできます。stringsはgrepコマ

ンドと組み合わせると便利です。下の例ではignoringが含まれるエラーメッセージを検索しています。

```
% cat /bin/ls | strings | grep ignoring
ignoring invalid tab size in environment variable TABSIZE: %s
ignoring invalid width in environment variable COLUMNS: %s
ignoring invalid value of environment variable QUOTING_STYLE: %s
```

「-tx」オプションを指定すると文字列の位置を16進数で表示することができます。10進数にしたい場合は-td、8進数の場合は-toを指定します。その他のオプションについてはman stringsでマニュアルを参照してください。

```
% strings -tx /bin/ls | head -5
  134 /lib/ld-linux.so.2
  b51 librt.so.1
  b5c clock_gettime
  b6a _Jv_RegisterClasses
  b7e __gmon_start__
```

文字列の判定

stringsはデフォルトでは「ASCIIの表示可能な文字(7ビット表現)で構成される4バイト以上の表示可能な文字列」というルールで文字列の判定を行います。このため、UTF-8でプログラムに埋め込んだ日本語の文字列(8ビット表現)はデフォルトでは表示されません。

```
const char *p = "日本語のメッセージです";
```

UTF-8の文字列を表示させるにはstringsに-eSオプションを渡します。-eはエンコーディング、Sは8ビット、という意味です。ただし、-eSを指定するとゴミのバイト列もたくさん引っかかってしまうので、-nオプションで表示に必要なバイト数(デフォルトは4)を増やすなどして対処する必要があります。

オブジェクトファイルの処理

stringsは内部的にBFDライブラリが解釈可能なオブジェクトファイル(実行ファイルやライブラリなど)かを判断して、オブジェクトファイルの場合はデータセクションのみを文字列抽出の対象とします。これはCのプログラムに含まれる、以下のような文字列は通常、データセクションに含まれるためです。

```
const char *p = "hello, world";
```

強制的にファイル全体を対象とする場合は-a、または-を指定します。ファイルがオブジェ

クトファイルでない場合はファイル全体が対象となります。また、標準入力から読み込んだ場合、BFDによる判定が行えないため、この場合もファイル全体が対象となります。システムの標準以外のオブジェクトファイルのフォーマットを指定するには-Tオプションを利用します。

まとめ

本 Hack では strings の使い方と仕組みについて紹介しました。strings はエラーメッセージの検索など、プログラムの簡単な解析に用いることができます。覚えておいて損はないツールです。

—— Satoru Takabayashi

HACK #14 c++filt で C++ のシンボルをデマングルする

本HackではC++のシンボルをコマンドラインからデマングルする方法としてnm --demangle と c++filt を紹介します。

C++ コンパイラはシンボルが一意の名前を持つように名前マングル（name mangling）と呼ばれる処理を行います。本 Hack ではコマンドラインから C++ のシンボルをデマングル（demangle）する方法を紹介します。実行時にデマングルする方法については「**[Hack #68] C++ のシンボルを実行時にデマングルする**」を参照してください。

nm の使い方

C++のオブジェクトファイルにnmをかけると、デフォルトではマングルされた読みづらい形式でシンボルが出力されます。

```
% nm foo.o
00000000 T _Z3fooi
```

これを読みやすくするにはパイプで c++filt を用いるか、nm に --demangle オプションを渡します。c++filt は nm の出力に限らず、汎用的なフィルタとして使えるので便利です。

```
% nm foo.o | c++filt
00000000 T foo(int)

% nm --demangle foo.o
00000000 T foo(int)
```

まとめ

本HackではC++のシンボルをコマンドラインからデマングルする方法としてnm --demangleとc++filtを紹介しました。nmもc++filtもGNU Binutilsに含まれるツールです。

—— Satoru Takabayashi

HACK #15 addr2lineでアドレスからファイル名と行番号を取得する

addr2lineはアドレスからファイル名と行番号を取得するツールです。主にデバッグ用途に使います。

addr2lineの使い方

addr2lineはデバッグ情報を利用してファイル名と行番号の情報を取得します。このため、プログラムはあらかじめデバッグ情報付きでコンパイルしておく必要があります。GCCでは-gオプションを指定します。

次のようなプログラムtest.cがあるとします。

```
#include <stdio.h>
void func() {
}

int main() {
    printf("%p\n", &func);
    return 0;
}
```

これをデバッグオプション付きでコンパイルします。実行すると関数funcのアドレスが表示されます。

```
% gcc -g test.c
% ./a.out
0x8048364
```

addr2lineを使うと、このアドレスから対応するファイル名と行番号を取得することができます。-eオプションで、対象の実行ファイルを指定しています。

```
% addr2line -e a.out 0x8048364
/tmp/test.c:2
```

確かにtest.cの2行目と表示されました。-fオプションを付けると関数名もわかります。

```
% addr2line -f -e a.out 0x8048364
func
/tmp/test.c:2
```

addr2lineに対し、標準入力からアドレスを渡すこともできます。これは、複数のアドレスをまとめて処理するときなどに便利です。

addr2line の仕組み

addr2lineは、デバッグ情報の取得にBFDを用いています。addr2lineと同様の処理を自分自身で行う方法は「**[Hack #67] libbfd でシンボルの一覧を取得する**」で紹介されています。

まとめ

本Hackでは、addr2lineを使ってアドレスからファイル名と行番号を取得する方法を紹介しました。デバッグに役立つノウハウではないかと思います。

—— Satoru Takabayashi

HACK #16 strip でオブジェクトファイルからシンボルを削除する

本Hackではstripの使い方を紹介します。

stripはオブジェクトファイルからシンボルを削除するツールです。通常、ビルドの完了した実行ファイルやライブラリから不要なシンボルを削除するのに使います。

strip の使い方

stripの使い方は簡単です。基本的にはシンボルを削除したいオブジェクトファイルを引数に指定するだけです。testという実行ファイルからシンボルを削除したい場合は次のように実行します。

```
% strip test
```

順を追って、例を見てみましょう。

```
void test() {}
int main() {
    return 0;
}
```

上のようなtest.cがあるとき、gccで普通にコンパイルすると、出来上がったtestにはシンボルの情報が含まれています。

```
% gcc -o test test.c
% nm test | grep test
08048334 T test
```

このとき、testは11,356バイトになりました。

testに対してstripを実行すると、シンボルが削除されます。

```
% strip test
% nm test
nm: test: シンボルがありません
```

シンボルの削除後は2,796バイトになりました。ほとんど空っぽのプログラムなのになぜ約8,000バイトものシンボルが含まれていたかというと、実はtestの大半は/usr/lib/crt*.oというオブジェクトファイルで占められていたからです。これらをリンクしない方法については「**[Hack #25] glibcを使わないでHello Worldを書く**」を参照してください。

stripの使い方のコツ

stripにはさまざまなオプションがありますが、とりわけ便利なのは-dオプションです。-dオプションを使うと、デバッグ用の情報(ファイル名や行番号など)だけを削除し、関数名などの通常のシンボルは残します。gdbでデバッグするときなどに関数名が残っていれば、ファイル名や行番号がわからなくてもかなり役に立ちます。ファイルのサイズは減らしたいけどデバッグに役立つ情報はある程度残したい、というときに便利です。

「-R」オプションは任意のセクションを問答無用に削除するオプションです。strip -R .text programなどと実行すると、プログラムのテキストセクション(コードの部分)がごっそり削除され、まったく動かなくなります。

また、.oおよび.aファイルに対してstripを適用すると、他のオブジェクトファイルとのリンクが事実上できなくなります。これはリンカがシンボルを頼りにしているためです。.oと.aファイルからシンボルを消すのはやめましょう。

なお、出荷する製品版のバイナリはstripしておいて、開発者側にデバッグ情報付きのバイナリを残しておくという方法をとることもできます。このようにしておくと、ユーザの環境で発生したコアファイルを開発環境でデバッグすることができます。

stripの実装

stripはBFDライブラリを用いて実装されています。BFDのAPIを駆使して、オブジェクト

ファイルの操作を行っています。Binutilsのソースコードを見るとobjcopyとコードが共通化されています。実はobjcopyでも-strip-*オプションを使うとstripと同様の処理を行えます。

まとめ

本Hackではstripの使い方を紹介しました。実行ファイルを小さくしたいという場面はディスク容量の豊富なPCではほとんどないかもしれませんが、容量の限られている環境にプログラムをインストールする際や、プログラムをネットワーク越しにコピーして実行するといった際などに使うと便利なツールです。

—— Satoru Takabayashi

HACK #17 arで静的ライブラリを操作する

arコマンドは静的ライブラリを作成、閲覧、編集、展開するためのコマンドです。本Hackではこのarコマンドを紹介します。

arの使い方

実はarコマンドの作成するアーカイブは静的ライブラリに限らず、tar(1)などと同様、汎用的な非圧縮のアーカイバとして使用することができます。しかし、通常は静的ライブラリを操作するために使用されます。

アーカイブを作成する場合は、ar rcus libhoge.a foo.o bar.o baz.o ...などとすればよいでしょう。オプションのrは新規なら挿入、既存なら置換を表しており、cはlibhoge.aが存在しない時の警告を抑制するコマンド、uはタイムスタンプを見て新しいものを置換するコマンド、sはranlib(1)と同等の処理を実行して書庫インデックスを作成するオプションです。インデックスを作成しないと、リンクが低速になったり、環境によってはエラーになったりします。このインデックスはnm -sで閲覧することができます。

アーカイブを閲覧する場合は、ar tv libhoge.aなどとするとよいでしょう。tは閲覧のコマンドで、vは饒舌モードを表します。これらによってファイルサイズや更新時刻などの情報も見ることができます。

アーカイブを展開する場合はar xv libhoge.aなどとします。vを付けることによって展開したファイルを確認しながら展開することができます。

arは普通に静的ライブラリを作成する場合だけでなく、静的ライブラリの動作を部分的に差し替えたい場合にも使用することができます。全ライブラリコードがオープンになっていないなどの理由で一から静的ライブラリを再作成できない場合に便利なテクニックだと思います。

arは他にもアーカイブ内の移動、削除など、アーカイブを操作するコマンドを持っていま

すが、ここで示したコマンドで、たいていの場合は問題ないでしょう。必要な場合は、適宜 man やヘルプで調べて下さい。

まとめ

本 Hack では、静的ライブラリを操作するツールである ar を紹介しました。

—— Shinichiro Hamaji

HACK #18 CとC++のプログラムをリンクするときの注意点

本Hackでは、CからC++の関数を呼び出す方法、逆にC++からCの関数を呼び出す方法を紹介します。

C++ からCで書かれた関数を呼び出したくなることはよくあります。あるいは逆に、Cから C++ で書かれた関数を呼び出したくなる事もたまにはあるでしょう。本 Hack では、それらを実現する方法と、その際の注意点を解説します。基礎的な Hack ですが、一から順に確認していきましょう。

C/C++ とシンボル名

次の dbg 関数を、C コンパイラ、C++ コンパイラの両方でコンパイルしてみます。

```
//
// dbg.c
//
#include <stdio.h>
void dbg(const char *s) {
  printf("Log: %s\n", s);
}
```

すると、生成されたオブジェクトに含まれるシンボル名は(例えば)次のようになるでしょう。

コンパイラ	シンボル名
C	dbg
C++	_Z3dbgPKc

Cコンパイラで関数をコンパイルすると、基本的には関数名がそのままシンボル名になります。環境によってはシンボル名が "dbg" ではなく "_dbg" になる場合もありますが、その程度の変化です。一方、C++コンパイラで関数をコンパイルした場合、**[Hack #14]**で解説しているように、関数の所属する名前空間の情報や、関数の引数の型情報がシンボルに含まれるようになります。

C++ から C の関数を呼び出す

さて、dbg.c を C コンパイラでコンパイルし、それを C++ で書かれた関数から呼び出してみましょう。次の sample.cpp を用意します。

```
//
// sample.cpp
//
extern "C" void dbg(const char *s);
int main() {
  dbg("foo");
  return 0;
}
```

これを、次のようにコンパイル、リンクすると、正常に実行ファイルが生成されます。無事、C++ の関数から C の関数を呼び出すことができました。

```
% gcc -Wall -c dbg.c
% g++ -Wall -c sample.cpp
% g++ -o sample dbg.o sample.o
% ./sample
Log: foo
```

sample.cpp の 4 行目、extern "C" がポイントです。extern "C" の有無で sample.o の内容がどう変化するか見てみましょう(nmコマンドの使い方は、「**[Hack #12] nmでオブジェクトファイルに含まれるシンボルをチェックする**」を参照してください)。

(1) extern "C" あり

```
% nm sample.o
         U __gxx_personality_v0
00000000 T main
         U dbg
```

(2) extern "C" なし

```
% nm sample.o
         U __gxx_personality_v0
00000000 T main
         U _Z3dbgPKc
```

(2)では、sample.o は "_Z3dbgPKc" というシンボルを参照していますが、dbg.o に含まれるシンボルは単なる"dbg"であり、"_Z3dbgPKc"はどこにも存在しません。これにより、extern "C" を付けないと、次のようにリンクに失敗する結果となります。

```
% g++ -o sample dbg.o sample.o
sample.o(.text+0x25): In function `main':
: undefined reference to `dbg(char const*)'
```

このように、C++からCの関数を呼び出す際には、extern "C"が重要な役割を果たします。

注意：extern "C" すると引数の型の一致が検査されない

ここで、sample.cpp を次のように書き換えてみます。

```
//
// sample.cpp
//
extern "C" void dbg(int i); // 誤った関数プロトタイプ宣言
int main() {
  dbg(1);
  return 0;
}
```

このファイルをコンパイル、リンクしてみると、なんとリンクに成功しています。もちろん、出来上がった実行ファイルは正常には動作しません。

```
% g++ -Wall -c sample.cpp
% g++ -o sample dbg.o sample.o
% ./sample
Segmentation fault
```

何故リンクに成功してしまうのでしょう？　それは、sample.oが参照しているのはあくまで、型情報を含まないシンボル"dbg"であり、それはdbg.oに含まれるシンボル名と一致するからです。

C++を使い慣れているとつい「リンクが成功すれば間違った型で関数が呼ばれることはない」と考えてしまいがちですが、C++であってもextern "C"を使用している部分に関してはそのかぎりではありません。安全性がC言語相当にまで低下してしまいます。この現象を未然に防ぐには、C言語側で、C++からも利用できるヘッダファイルを用意しておくのが良いでしょう。たとえば、次のようなdbg.hを用意するという方法が定番です。

```
//
// dbg.h
//
#ifdef __cplusplus
  extern "C" {
#endif
    void dbg(const char *s);
#ifdef __cplusplus
  }
#endif
```

C から C++ の関数を呼び出す

今度は逆に、C から C++ の関数を呼び出してみましょう。「最大公約数を求める関数」を、Boost C++ Library（http://www.boost.org/）という著名なライブラリを用いて C++ で簡単に実装し、その関数を C から呼び出してみます。ここで、ポイントは次の 2 つです。

- C++ 側の関数を C リンケージ（extern "C"）でコンパイルする
- リンクは、gcc ではなく g++ コマンドで行う

これに従うと、C++ 側の実装は次のようになります。

```
//
// gcd.h
//
#ifdef __cplusplus
extern "C"
#endif
int gcd(int v1, int v2);

//
// gcd.cpp
//
#include <boost/math/common_factor.hpp>
#include "gcd.h"

extern "C" {
  int gcd(int v1, int v2) {
    return boost::math::gcd(v1, v2);
  }
}
```

C++ の関数を呼び出す側には、特に変わったところはありません。

```
//
// sample.c
//
#include <stdio.h>
#include "gcd.h"

int main() {
  printf("%d と %d の最大公約数は %d です\n", 14, 35, gcd(14, 35));
  return 0;
}
```

では、コンパイルして実行してみましょう。

```
$ g++ -Wall -c gcd.cpp
$ gcc -Wall -c sample.c
$ g++ -o sample sample.o gcd.o
$ ./sample
14と35の最大公約数は7です
```

無事に動作しました。C から C++ の関数を呼び出すことができています。

注意：C++ の関数は例外を C の関数側に漏らしてはならない

C 言語から呼ばれる C++ の関数を書く際には、C++ の例外が C 言語側に到達しないように注意してください。C++ の例外が C で書かれた関数に到達した場合の動作は、C++ の規格では明確に決まっておらず、例えば GCC ではプロセスが異常終了してしまいます。

C++ 関数で明示的に例外を throw しないよう心掛けるのはもちろん、以下の点などにも充分に注意しましょう。

- new 演算子が std::bad_alloc 例外を throw する可能性
- std::vector のメンバ関数 at が、std::out_of_range 例外を throw する可能性

次のように、C++ の関数全体を try/catch で囲んでおくのもよいアイデアです。

```
extern "C" {
  int cpp_func() try {
    // ... 例外をthrowする可能性のある処理 ...
    return 0;  // 成功
  } catch(...) {
    return -1; // 失敗
  }
}
```

注意：関数ポインタを扱う C の関数に注意

Cの関数の中をC++の例外が通過するのを抑制できないケースがあります。標準Cのqsort関数が、次のように C++ から使われた場合を考えてみてください。

```
//
// sample2.cpp
//
#include <cstdlib>
using namespace std;

// qsortの比較関数
int compar(const void *, const void *) {
  throw -1;
}
```

```
int main() {
  int array[] = {3, 2, 1};
  try {
    qsort(array, 3, sizeof(int), compar);
  } catch(...) {
    return 1;
  }
  return 0;
}
```

qsort関数に渡した比較関数comparが例外をthrowするため、qsort関数の中をC++の例外が通過してしまいます。GCCを使っている場合で、このようなケースでもプロセスを異常終了させないためには、下のようにして、qsort関数を"-fexceptions"付きでコンパイルしなければなりません。

```
% gcc -fexceptions qsort.c
```

一般に、関数ポインタを扱うCの関数は、-fexceptionsオプション付きでコンパイルしておくのが無難といえます。glibcのMakefileを見ると、そういう関数(qsort、bsearchほか)はこのオプション付きでコンパイルされるようになっています。

まとめ

本Hackでは、CからC++の関数を呼び出す方法と、逆にC++からCの関数を呼び出す方法について解説しました。また、これらを行う際に注意しなければならない点として、以下の点を解説しました。

- extern "C"の扱い方
- C++における例外の扱い方

後者に関係する、GCCの-fexceptionsオプションは、あまり知られていないオプションながら、関数ポインタを扱うC関数をコンパイルする際には重要な役割を果たしますので、ぜひ覚えておいてください。

なお、関数呼び出しに関するさらに高度なHackは、「**[Hack #69] ffcallでシグネチャを動的に決めて関数を呼ぶ**」などで紹介されています。

—— Yusuke Sato

HACK #19 リンク時のシンボルの衝突に注意する

本Hackでは静的リンクおよび動的リンク時に発生するシンボルの衝突の問題と回避策を紹介します。

同名シンボルの衝突

CやC++のプログラムで同じ名前のグローバルなシンボルが2つ以上存在するとどうなるでしょうか。

.o ファイルをまとめてリンクする場合

まず、次のようなファイルa.cがあります。a.cではグローバルな関数func()を定義しています。

```
#include <stdio.h>
void func() {
    printf("func() in a.c\n");
}
```

次に、b.cでも同様にfunc()を定義しています。a.cのものとよく似ていますが、printfで表示されるメッセージは異なります。

```
#include <stdio.h>
void func() {
    printf("func() in b.c\n");
}
```

最後に、main.cではfunc()を呼び出しています。

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

これらの3つのファイルをそれぞれコンパイルして静的にリンクしようとすると、func()が複数あるため、リンカがエラーを検出します。このエラーのおかげで、予期せぬfunc()が呼ばれるという事態を避けられます。

```
% gcc -c a.c
% gcc -c b.c
% gcc -c main.c
% gcc -o main a.o b.o main.o
b.o(.text+0x0): In function `func':
: multiple definition of `func'
```

```
a.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

また、a.oとb.oから共有ライブラリlibfoo.soを作る場合もリンカがシンボルの衝突を検出してエラーが発生します。

```
% gcc -fPIC -c a.c
% gcc -fPIC -c b.c
% gcc -shared -o libfoo.so a.o b.o
b.o(.text+0x0): In function `func':
: multiple definition of `func'
a.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

ライブラリを作ってリンクする場合

一方、同じソースコードa.cとb.cからarを使って静的ライブラリを作るとリンク時のエラーは発生しません。リンカと違い、オブジェクトファイルのアーカイバであるarはシンボルの衝突の検出を行わないためです。

```
% gcc -c a.c
% gcc -c b.c
% gcc -c c.c
% ar cr libfoo.a a.o b.o
% gcc main.o libfoo.a
```

実行すると、a.cのfunc()が呼び出されます。これは、リンク時にlibfoo.aの中の2つのfunc()のうち、a.cのものがb.cのものより先に見つかるからです。

```
% ./a.out
func() in a.c
```

静的ライブラリlibfoo.aをarで作るときの引数の順序をa.o b.oからb.o a.oに変更すると、b.o内のfunc()が使われます。

また、a.cとb.cに対してそれぞれ静的ライブラリを作った場合も、リンク時のエラーは発生しません。このときgccに渡すliba.aとlibb.aの順序を逆にするとb.cのfunc()が呼ばれます。

```
% ar cr liba.a a.o  # liba.a を作成
% ar cr libb.a b.o  # libb.a を作成
% gcc main.o liba.a libb.a
# ./a.out
func() in a.c
```

同様に、a.cとb.cからそれぞれ動的共有オブジェクト(共有ライブラリ)a.soとb.soを作ってリンクするとエラーは発生しません。下の例の場合、a.soの中のfunc()が呼び出されます。

```
% gcc -fPIC -shared -o a.so a.c
% gcc -fPIC -shared -o b.so b.c
% gcc -fPIC -shared -o main.so main.c
% gcc -o main-shared ./a.so ./b.so ./main.so
```

リンク時のコマンドライン引数のa.soとb.soの順序を逆にすると、b.soのfunc()が先に見つかるようになります。

動的リンクの際には、基本的に初めに見つかったシンボルの定義が使われます。GNU C Libraryの主要開発メンバーであるUlrich Drepper氏による「How to Write Shared Libraries」(http://people.redhat.com/drepper/dsohowto.pdf)に次のような記述があります。

> スコープ内に同じシンボルの定義が2つ以上含まれていても構わない。シンボルルックアップのアルゴリズムは単純に最初に見つかった定義を拾うだけだ。……このコンセプトは非常に強力であり……その一例はLD_PRELOADの機能の利用である……

LD_PRELOADについては「**[Hack #60] LD_PRELOADで共有ライブラリを差し換える**」を参照してください。

このように、静的ライブラリ、共有ライブラリともに、別々のライブラリに同名のシンボルの定義が含まれていてもプログラムは正常にリンク、実行できます。この動作は「予期せぬ関数が呼ばれて謎の挙動をする」といったバグが発生する可能性があるため注意が必要です。

C++と同名クラス

同一シンボルの問題は、C++では予期せぬメンバ関数が呼ばれるという問題を引き起こします。

消費税を計算するa.{h,cpp}というソースコードと、それを利用するmain.cppがあるとします。

a.h

```
class Tax {
public:
    int tax(int price);
    Tax();
private:
```

```
    double consumption_tax_;
};
```

a.cpp

```
#include "a.h"
Tax::Tax() : consumption_tax_(1.05) {}

int Tax::tax(int price) {
    return price * consumption_tax_;
}
```

main.cpp

```
#include <iostream>
#include "a.h"
int main() {
    Tax tax;
    int apple_price = 100;
    std::cout << "apple: " << tax.tax(apple_price) << std::endl;
    return 0;
}
```

これらのソースコードをコンパイル、リンクして実行するとapple: 105というメッセージが表示されます。

```
% g++ -fPIC -shared -o a.so a.cpp
% g++ -fPIC -shared -o main.so main.cpp
% g++ -o main-shared ./a.so ./main.so
% ./main-shared
apple: 105
```

ここで、a.{h,cpp}とはまったく独立に開発されたb.cppが存在し、同名のクラスTaxを実装していたとします。b.cpp内のTaxクラスの用途はa.cppとはまったく異なります。

```
class Tax {
public:
    int deduct(int income);
    Tax();
private:
    double deduction_rate_;
};

Tax::Tax() : deduction_rate_(0.1) {}

int Tax::deduct(int income) {
    return int(income * (1.0 - deduction_rate_));
}
```

そして、b.cppがb.soとしてコンパイルされ、運悪くb.so、a.so、main.soの順序でリンクされると、厄介なことが起きます。

```
% g++ -o main-shared2 ./b.so ./a.so ./main.so
% ./main-shared2
apple: 10
```

今回の実行結果はapple: 10です。消費税込みの価格を計算するつもりが、90%ディスカウントになってしまいました。

これは、Taxオブジェクトのコンストラクタとしてb.so内のTax::Tax()が呼ばれ、Taxオブジェクトのメンバ関数としてa.so内のTax::tax()が呼ばれているためです。つまり、ここでは予期せぬコンストラクタが呼ばれてしまったことがバグの原因になっています。

同名クラスの衝突によるこのような問題は小さなプログラムではまず発生しませんが、プログラムが巨大になるにつれて発生しやすくなると思います。namespaceを使うなどの方法で衝突をさける必要があるでしょう。

シンボルを1つのファイル内に閉じ込めるだけなら無名namespaceが有効です。たとえばb.cppの全体をnamespace { ... }で囲めば、上記の問題は発生しません。

weakシンボル

最初のところで、.oファイルをまとめてリンクする際には同名シンボルの衝突は検出されると書きました。しかし、weakシンボルが存在すると話は変わります。

次のようなプログラムmain.cppを考えます。

```
#include <iostream>
class Foo {
public:
    Foo(int x) : x_(x) {}
    void func() {
        std::cout << x_ << std::endl;
    }
private:
    int x_;
};

int main () {
    Foo foo(256);
    foo.func();
    return 0;
}
```

このプログラムをコンパイルして実行すると256と表示されます。

```
% g++ -c main.cpp
% g++ -o main main.o
% ./main
256
```

ここで、main.cpp とはまったく独立に開発された次のような a.cpp が存在します。a.cpp 内でもクラス Foo を実装しています。

```
#include <iostream>
class Foo {
public:
    Foo(int x);
private:
    int x_;
};

Foo::Foo(int x) : x_(x * x) {}
```

そして、a.cppをコンパイルしたa.oが運悪くmainにリンクされると大変厄介なことが起きます。

```
% g++ -c a.cpp
% g++ -o main main.o a.o
% ./main
65536
```

今回の実行結果は65536です。256ではなく、その二乗が表示されてしまいました。このとき、リンクの順は main.o, a.o でも a.o, main.o でもどちらでも結果は変わりません。

main.cpp 内で定義されたコンストラクタ Foo::Foo(int)ではなく a.cpp 内の Foo::Foo(int)が使われていることは明らかですが、なぜこのようなことが起きるのでしょうか。

weak シンボルとは

原因は、main.o に含まれる Foo::Foo(int)が weak シンボルであり、a.o に含まれる Foo::Foo(int)が非 weak シンボルであることにあります。以下の nm の出力で W と表示されているのが weak シンボルの印です。

```
% nm --demangle main.o | grep Foo::
00000000 W Foo::func()
00000000 W Foo::Foo(int)
% nm --demangle a.o | grep Foo::
00000012 T Foo::Foo(int)
00000000 T Foo::Foo(int)
```

weak シンボルについては『Linkers & Loaders』（John R. Levine 著、榊原一矢監訳、ポジ

ティブエッジ訳、オーム社)の6章に以下の解説があります。

> ELFでは、weak参照のほかにweak定義というもう1つのweakシンボルを追加している。weak定義は、通常の定義が存在しない場合に大域シンボルを定義する。通常の定義が存在する場合は、weak定義は無視される。

つまり、a.oに通常の(非weakの)定義が存在するためにmain.oのFoo::Foo(int)のweak定義は無視されているということです。

ひとまず、この問題を回避するには、無名namespaceを用いてリンケージリークを防ぐのが効果的です。具体的にはmain.cppのclass Foo { ... }の周りをnamespace { ... }で囲みます。

weak定義と重複コードの除去

ところで、なぜmain.oのFoo::Foo(int)およびFoo::func()はweak定義になっているのでしょうか。これは、g++はインライン関数をweak定義としてコンパイルするためです(クラス定義内の関数定義はC++の規格によりインライン関数とみなされます)。そして、weak定義は.oファイル内の.gnu.linkonce.t.*という名前のセクションに配置されます。

.gnu.linkonce.t.*はリンク時の重複コードの除去に使われるセクションです。クラス定義は多くの場合、.hファイル内に記述され、.hファイルは複数の.cppファイルに#includeされます。そのため、各.oファイルに含まれたweak定義をリンク時に1つにまとめる必要があります。インライン関数がweak定義になるのはこのためです。

まとめ

本Hackでは静的リンクおよび動的リンク時に発生するシンボルの衝突の問題と回避策について紹介しました。通常、リンカの動作などはあまり気にしなくてもプログラムを書けますが、上記のような問題が発生した場合は、リンカの知識の有無によってバグ修正にかかる時間が大幅に変わってくると思います。

—— Satoru Takabayashi

HACK #20 GNU/Linuxの共有ライブラリを作るときPICでコンパイルするのはなぜか

本Hackでは、共有ライブラリを作成する際に、なぜPICでコンパイルする必要があるのか調べてみます。

通常、GNU/Linuxの共有ライブラリを作るときは各.cファイルをPIC(Position Indepen

dent Code）となるようコンパイルします。しかし、実はPICでコンパイルしなくても共有ライブラリは作れます。それではPICにする意味はあるのでしょうか。

さっそく実験してみます。

```
void func() {
    printf("");
    printf("");
    printf("");
}
```

PICでコンパイルするにはgccに-fpicまたは-fPICを渡します。-fpicの方が小さく高速なコードを生成する可能性がありますが、プロセッサによっては-fpicで生成できるGOT（Global Offset Table）のサイズに制限があります。一方、-fPICはどのプロセッサでも安心して使えます。ここでは-fPICを用います（x86では-fpicも-fPICも同じです）。

```
% gcc -o fpic-no-pic.s -S fpic.c
% gcc -fPIC -o fpic-pic.s -S fpic.c
```

上のように生成したアセンブラのソースを見ると、PIC版はprintfをPLT（Procedure Linkage Table）経由で呼んでいることがわかります。

```
% grep printf fpic-no-pic.s
        call    printf
        call    printf
        call    printf
% grep printf fpic-pic.s
        call    printf@PLT
        call    printf@PLT
        call    printf@PLT
```

次に、共有ライブラリを作ります。

```
% gcc -shared -o fpic-no-pic.so fpic.c
% gcc -shared -fPIC -o fpic-pic.so fpic.c
```

これらの共有ライブラリの動的セクション（dynamic section）をreadelfで見ると、非PIC版にはTEXTRELというエントリがあり（テキスト内の再配置が必要）、さらにRELCOUNT（再配置の数）が5と、PIC版より3つ多くなっています。3つ多いのはprintf()の呼び出しを3回行っているためです。

```
% readelf -d fpic-no-pic.so|egrep 'TEXTREL|RELCOUNT'
 0x00000016 (TEXTREL)                    0x0
 0x6ffffffa (RELCOUNT)                   5
```

```
% readelf -d fpic-pic.so|egrep 'TEXTREL|RELCOUNT'
 0x6ffffffa (RELCOUNT)                   2
```

PIC版のRELCOUNTが0でないのはgccがデフォルトで使うスタートアップファイルに含まれるコードが原因です。gccに-nostartfilesオプションを渡すと0になります。

PICと非PICの共有ライブラリの性能比較

上の例では非PIC版は実行時(動的リンク時)に5つのアドレスの再配置が必要と書きました。では、再配置の数が膨大に増えたらどうなるでしょうか。

次のシェルスクリプトを実行すると、printf()の呼び出しを1000万回含む共有ライブラリを非PIC版とPIC版で作り、それらをリンクした実行ファイルfpic-no-picとfpic-picを作ります。

```
#! /bin/sh
rm -f *.o *.so
num=1000
for i in `seq $num`; do
  echo "void func$i() {" > fpic$i.c
  ruby -e "10000.times { puts 'printf(\"\");' }" >> fpic$i.c
  echo "}" >> fpic$i.c
  gcc -o fpic-no-pic$i.o -c fpic$i.c
  gcc -fPIC -o fpic-pic$i.o -c fpic$i.c
done
gcc -o fpic-no-pic.so -shared fpic-no-pic*.o
gcc -o fpic-pic.so -shared fpic-pic*.o
echo "int main() { return 0; }" > fpic-main.c
gcc -o fpic-no-pic fpic-main.c ./fpic-no-pic.so
gcc -o fpic-pic fpic-main.c ./fpic-pic.so
```

できあがった実行ファイルの実行結果は以下の通りです。非PIC版が初回2.15秒、2回目以降約0.55秒かかっているのに対し、PIC版は初回0.02秒、2回目以降は0.00秒となっています。

```
% repeat 3 time ./fpic-no-pic
2.15s total : 0.29s user 0.48s system 35% cpu
0.56s total : 0.25s user 0.31s system 99% cpu
0.55s total : 0.30s user 0.25s system 99% cpu

% repeat 3 time ./fpic-pic
0.02s total : 0.00s user 0.00s system 0% cpu
0.00s total : 0.00s user 0.01s system 317% cpu
0.00s total : 0.00s user 0.00s system 0% cpu
```

main()の中身は空ですから、非PIC版は動的リンク時の再配置に2.15～0.55秒を要してい

ることがわかります。実行環境は、Xeon 2.8 GHz + Debian GNU/Linux sarge + GCC 3.3.5です。

非PIC版のデメリットは実行時の再配置に時間がかかるだけではありません。再配置が必要な部分のコードを書き換えるために、「テキストセグメント内の再配置が必要なページをロード→書き換え→copy on write発生→他のプロセスとテキストが共有できない」という事態が発生します。つまりこれではテキスト(プログラムのコード)を他のプロセスと共有するという「共有」ライブラリの主要なメリットが消失してしまいます。

ところで、非PIC版の`fpic-no-pic.so`とPIC版の`pic.so`のサイズを比べると、前者は268MB、後者は134MBと大きく異なりました。`readelf -S`でセクションヘッダを見ると、次のような違いがありました。

	.rel.dyn	.text
非 PIC	152MB	114MB
PIC	0MB	133MB

非 PIC 版はコード(`.text`)は PIC 版より小さくなっていますが、再配置に必要な情報(`.rel.dyn`)が膨大な容量を占めています。

まとめ

本 Hack では、共有ライブラリを作成する際に、なぜ PIC でコンパイルする必要があるのか調べてみました。非PICの共有ライブラリを作成することは可能ですが、実行時の再配置に時間がかかり、さらに他のプロセスとコード(`.text`)が共有できないという大きなデメリットがあります。共有ライブラリを作成する際には`.c`ファイルをPICでコンパイルするようにしましょう。

—— Satoru Takabayashi

HACK #21 statifierで動的リンクの実行ファイルを擬似的に静的リンクにする

statifierを使うと、動的リンクされた実行ファイルと共有ライブラリを1つのファイルにまとめることができます。

statifierは動的リンクされた実行ファイルと共有ライブラリを1つのファイルにまとめるための GNU/Linux 用のツールです。動的リンクされた実行ファイルを別のホストにコピーして実行したい、というときなどに使えます。

statifier は原稿執筆時点では Debian パッケージになっていないため、Debian で使うには

ソースコードから make && make install します。

```
% tar zxf statifier-1.6.7.tar.gz
% cd statifier-1.6.7
% make
% sudo make install
```

statifier の使い方

使い方は簡単です。ターゲットの実行ファイルと新しいファイル名をコマンドオプションで指定して statifier を実行するだけで OK です。たとえば /usr/bin/php と共有ライブラリをまとめて php2 という 1 つのファイルにまとめるには次のように実行します。

```
% statifier /usr/bin/php php2
```

このようにして作成した php2 ファイルは別のホストにコピーして実行できます。試しにPHPがインストールされていないホストに/usr/bin/phpとphp2をコピーして実行してみたところ、php2 は無事に実行できました。/usr/bin/php をコピーしただけのファイルは、ライブラリが足りないため、実行できません。

```
% scp /usr/bin/php php2 foo.example.com:
% slogin foo.example.com

% ./php -v
./php: error while loading shared libraries: libzzip-0.so.12: cannot open   shared
object file: No such file or directory

% ./php2 -v
PHP 4.3.10-16 (cli) (built: Aug 24 2005 20:25:01)
Copyright (c) 1997-2004 The PHP Group
Zend Engine v1.3.0, Copyright (c) 1998-2004 Zend Technologies

% ldd php |grep 'not found'   # いくつかの共有ライブラリが欠けている
        libzzip-0.so.12 => not found
        libgssapi_krb5.so.2 => not found
        libkrb5.so.3 => not found
        libk5crypto.so.3 => not found
```

/lib/libnss*.so 問題

glibcのシステムではDNSなどを用いた名前解決に/lib/libnss*.soを必要とします。これらの共有ライブラリは最初に名前解決を行うときにロードされるため、依存関係は静的にはわかりません。そのため、statifierは生成したファイルにlib/libnss*.soを含めることができません。

試しにgetentコマンドをstatifierで処理して実行してみると、やはり実行時に/lib/libnss*をロードしていました。もしstatifierで生成したバイナリを異なるLinuxマシンで実行する場合、/lib/libnss*のロードに失敗してプログラムの実行が異常終了する可能性があります。

```
% statifier /usr/bin/getent getent2
% LANG=C strace ./getent2 hosts >/dev/null 2>&1 |grep /lib
open("/lib/libnss_files.so.2", O_RDONLY) = 3
open("/lib/libnss_dns.so.2", O_RDONLY)  = 4
open("/lib/libresolv.so.2", O_RDONLY)   = 4
```

これらの共有ライブラリをバイナリに含めるにはstatifierの--setオプションを使います。LD_PRELOAD環境変数に、バイナリに含めたい共有ライブラリを空白区切りで設定すればOKです。

```
% statifier --set=LD_PRELOAD="/lib/libnss_files.so.2 /lib/libnss_dns.so.2 /lib/
libnss_dns.so.2" /usr/bin/getent getent3
```

iconv(3)も実行時に/usr/lib/gconv/*.soをロードするため、同様の問題が起きます。glibcのgettextは内部でiconvを呼び出しているため、gettext化したプログラムをstatifierで扱うときは必要な.soファイルを--set=LD_PRELOADを使って含める必要があります。

statifierの仕組み

statifierはELFとglibcの環境を前提とした作りになっています。ELFとglibcの環境では、動的リンクされたプログラムの実行時にld.so(/lib/ld-linux.so.2など)が必要な共有ライブラリのロードを行います。この処理は大きく次の2つにわかれます。

1. 動的リンクに必要な再配置などの処理をすべてのライブラリに対して行い、メモリ上にマップする
2. 各ライブラリ内の初期化関数を呼び、元のプログラムのエントリーポイントに処理を移す

statifierのアイディアは2の直前の時点でのメモリのスナップショットをとれば、単一ファイルで実行できるバイナリを作れるのではないか、というものです。ld.soの中では、1の処理を_dl_start()関数で行い、2の処理を_dl_start_user()関数で行っています。ld.soのこの動作を前提として、statifierがバイナリを作る際には次のような処理を行います。

- _dl_start_user()にブレークポイントを設定してターゲットのプログラムを実行する
- _dl_start_user()に到達した時点でどのようにオブジェクトがメモリにマップされているかを /proc/PID/maps を元に調べる
- マップされているそれぞれの領域を、ファイルにダンプして保存する
- 最後にそれらを連結して単一のバイナリにする

実際には、TLS使用の有無のチェックやレジスタの保存など、他にも各種の処理を行っていますが、基本的な流れは以上です。

生成されたバイナリの実行時には、レジスタの復元や変数のセットなどの前処理を行ったのちに、_dl_start_user()に飛び込んで処理を開始します。これは、さきほどの2の直前でとっておいたスナップショットを2から再開すると考えるとわかりやすいと思います。

前述の通り、statifierはELFとglibcの環境を前提とした作りとなっているため、将来的にはglibcのバージョンが上がってld.soの動作が変わると、statifierの仕組みも変わる可能性があります。

まとめ

statifierを使うと、動的リンクされた実行ファイルと共有ライブラリを1つのファイルにまとめることができます。statifierのサイトには仕組みを説明した文書があります。より詳しい仕組みを知りたい方はぜひ参照してください。

—— Satoru Takabayashi

3章

GNU プログラミング Hack

Hack #22-41

GNU/Linuxはその名前が示すように、GNUプロジェクトの成果がたくさん含まれています。実際、GNU/Linux は、Linux カーネルを除くと、C ライブラリ、コンパイラ、リンカ、デバッガといった重要なコンポーネントがすべて GNU プロジェクトの成果が使われているのです。本章では、GCC が提供する拡張機能を中心に、GNU のツールチェーンが提供する独自の機能を活用する方法を紹介します。GNU のツールチェーンを使いこなすことは、GNU/Linux 上で Binary Hack を行う上で最重要課題と言えるでしょう。

HACK #22 GCC の GNU 拡張入門

GCC の GNU 拡張機能の代表的なものを３つ(ビルトイン関数、アトリビュート、ラベルの参照)紹介します。

本 Hack では、GCC の GNU 拡張機能の代表的なものを紹介します。

ビルトイン関数

GCCでは標準にあるような関数のいくつかをビルトイン関数として用意しており、最適化によってはソースに書かれているものとは違うコードを生成する場合があります。例えばprintf(3)などの場合、次のような文字列リテラルを出力するコード、

```
printf("hello, world\n");
```

では実行時にprintf(3)のフォーマット文字列解析をするのは無駄なので、次のようなputs(3)の呼び出しコードが生成されます。

```
puts("hello, world");
```

基本的には同じ動作で、より高速に動作するようなコードになるので問題はありませんが、LD_PRELOAD などでオーバーライドして動作を変更させたい場合などに注意する必要があります。最適化でどのようなコードを生成しているかはgcc/builtins.cでプログラムされています。

その他に動作時の状況を知るビルトイン関数や、コンパイル時にヒントをあたえるようなビルトイン関数があります。

`void *__builtin_return_address(unsigned int LEVEL)`
: 関数のリターンアドレス、つまりその関数の現在の呼び出し元を返します。LEVELは呼び出し元をいくつたどるかを定数で指定しますが、通常0(現在の関数)を指定します。通常、x86の場合は%ebp + 4にあるポインタ値になります。

`void *__builtin_frame_address(unsigned int LEVEL)`
: 関数のフレームポインタ(ローカル変数やレジスタの保存領域を指すポインタ)を返します。LEVELは呼び出し元をいくつたどるかを定数で指定しますが、通常0(現在の関数)を指定します。通常、x86の場合は%ebpになります。

`int __builtin_types_compatible_p (TYPE1, TYPE2)`
: TYPE1とTYPE2がコンパチブルかどうかを調べます。通常はマクロでタイプによって適当な関数にディスパッチする時に使います。

`TYPE __builtin_choose_expr(CONST_EXP, EXP1, EXP2)`
: これはCONST_EXP ? EXP1 : EXP2と同じようなものですが、コンパイル時に、どちらにするかを決めてしまうところが異なります。__builtin_types_compatible_pをCONST_EXPに使って、適当な関数呼び出しを選ぶようなマクロを書く時などに使います。

`int __builtin_constant_p (EXP)`
: EXPが定数かどうかを判定します。引数が定数だとわかるときに最適化したコードを選んだほうがいい場合などに使います。

`long __builtin_expect(long EXP, long C)`
: EXPの値がCになることが多いという判断のもとで、ブランチの最適化をかけてほしい場合に使います。

`void __builtin_prefetch (const void *ADDR, int RW, int LOCALITY)`
: ADDRにあるデータをキャッシュにプリフェッチしたい時に使います。RWが1の時は近いうちに書き込みがあることを、0の時は近いうちに読み込みがあることを伝えます。LOCALITYは0～3の値で、0は使ったらすぐ不要になるデータ、3は使いはじめるとしばらく使い続けるデータであることを伝えます。

アトリビュート(__attribute__)

関数にアトリビュートをつけて宣言することで、関数に特別の意味をもたせたり関数の呼び出しの最適化を期待したりすることができます。

アトリビュートは次のようにして宣言します。

```
int foo(int n) __attribute__((アトリビュート));

int foo(int n)
{
  …
}
```

または次のように書くこともできます。

```
__attribute__((アトリビュート)) int foo(int n)
{
}
```

関数に対するアトリビュートには次のようなものがあります。

constructor
: mainが呼ばれる前や共有オブジェクトがロードされた時に実行すべき関数に使います。

destructor
: exitする前や共有オブジェクトがアンロードされる直前に実行すべき関数に使います。

cleanup
: auto 変数がスコープから消えてなくなる時に呼び出される関数を指定します。

section
: 特定のセクションにコードを配置します。

used
: どこからも呼び出しがない場合でもコードをかならず生成します。アセンブラコードからのみ呼ばれるような場合に使います。

weak
: weak シンボルなコードを生成します。

alias
: 別のシンボルへのエイリアスとします。通常 weak と共に使います。

visibility
シンボルの可視性を制御するのに使います。"default"、"hidden"(共有オブジェクトの外からは見えない)、"protected"(共有オブジェクト内の呼び出しはLD_PRELOADなどでもオーバーライドされない)、"internal"(関数ポインタなどを含めて共有オブジェクト外部からの呼び出しを行わない)などがあります。

stdcall
x86の呼び出し規約の指定で、引数に使ったスタックを呼び出された側でpopする場合に使います。

cdecl
x86の呼び出し規約の指定で、引数に使ったスタックを呼び出し側でpopする場合に使います。

fastcall
x86の呼び出し規約の指定で、最初の2つの引数を%ecx、%edxを使って呼び出します。

regparm
引数をいくつレジスタ渡しするかを制御します。

vector_size
変数のベクタサイズを指定します。

dllimport
Windows の dllimport です。

dllexport
Windows の dllexport です。

pure
グローバル変数と引数によってのみ返り値が決まり、副作用がない場合に使えます。場合によっては無駄な呼び出しが省かれる場合があります。

const
引数だけで返り値が決まり、副作用がない場合に使えます。

malloc
NULL 以外の返り値が他のポインタと共有されていない場合に使います。

noreturn
　exit(2)などのように戻ってこない関数に使います。

noinline
　インライン展開させたくない場合に使います。

always_inline
　最適化レベルが低くてもインライン展開します。

nothrow
　例外を投げない場合に使います。

format
　フォーマット文字列がprintf、scanf、strftime、strfmonのどのスタイルと同じかを示すことでフォーマット文字列と可変引数の対応のチェックを行うようになります。

format_arg
　どの引数がフォーマット文字列かを示します。

nonnull
　NULLにならないポインタ引数を示します。

unused
　使われない場合でも警告を出しません。

deprecated
　使われた時に警告を出します。

warn_unused_result
　返り値をチェックしていない時に警告を出します。

no_instrument_function
　-finstrument-functionsの場合でもプロファイル関数の呼び出しをしないようになります。

データに対するアトリビュートには次のようなものがあります。

aligned
　変数の領域のアライメントを制御します。

packed
structure内部などでアライメントによるパッディングを最小にします。

common
変数をコモン領域に配置します。

nocommon
変数コモン領域に配置しません。

shared
DLLを使うプロセスすべてで共有される変数に使います。

ラベルの参照

Cではそれほど使われませんが、gotoでジャンプする時にジャンプする先を指定するのにラベルを使います。

```
    goto error;
    ....
error:
    /* エラー処理 */
```

GCCでは、このラベルは&&で参照することでvoid *型の変数に代入することができます。

```
    void *label;
    label = &&error;
    ...
    goto *label;
    ....
error:
    ...
```

このようにラベルを変数に代入することができるので、変数の値によってジャンプ先を変更するようなコードは、ラベルへの参照を要素に持つ配列を使うなどして実現することができます。

ラベルを&&で参照した値はvoid *に代入できるようなポインタ型なので、引き算することによってオフセットを得ることができます。したがって次のようなコードを書くこともできます。

```
    static int labals[] = { &&label0 - &&label0, &&label1 - &&label0, ... };
    ....
    goto *(&&label0 + labels[i]);
    ....
label0:
```

```
    ....
label1:
    ....
```

まとめ

GCCは、標準のC99などに準拠するように努める一方、ソースコードを記述しやすくするような拡張をいくつか提供しています。GCCの拡張機能であると理解した上で、これらの機能を使うと便利な場合があります。例えばLinuxのカーネルなどはGCCの拡張機能を利用したコードが見られます。そのようなコードを読む場合にはGCCの拡張機能の理解が欠かせません。

—— Fumitoshi Ukai

HACK #23 GCC でインラインアセンブラを使う

GCC の asm 命令を使用することで C の中でインラインアセンブラを使うことができます。

本 Hack では、GCC でインラインアセンブラを使う方法について説明します。

変数にレジスタを割り当てる

ある変数を特定のレジスタに割り当てることができます。次のようにすると%esp、%ebpを C の変数 stack_pointer、frame_pointer で参照することができるようになります。

```
register void *stack_pointer asm ("%esp");
register void *frame_pointer asm ("%ebp");
```

インラインアセンブラ

GNU/Linux の <string.h> では次のようにインラインアセンブラで高速な strcpy()などを定義しています。

```
26  #define __HAVE_ARCH_STRCPY
27  static inline char * strcpy(char * dest,const char * src)
28  {
29  int d0, d1, d2;
30  __asm__ __volatile__(
31          "1:\tlodsb\n\t"
32          "stosb\n\t"
33          "testb %%al,%%al\n\t"
34          "jne 1b"
35          : "=&S" (d0), "=&D" (d1), "=&a" (d2)
36          :"0" (src),"1" (dest) : "memory");
```

```
37  return dest;
38  }
```

__asm__ 命令は「:」で区切った次のような形式になっています。

```
__asm__ ("アセンブラテンプレート"
        : 出力オペランドの設定
        : 入力オペランドの設定
        : アセンブラの実行で変更されてしまうもの)
```

上のstrcpy()の場合、次のようになります。

アセンブラテンプレート

```
1:  lodsb
    stosb
    testb %al,%al
    jne 1b
```

出力オペランド

```
"=&S" (d0), "=&D" (d1), "=&a" (d2)
```

入力オペランド

```
"0" (src),"1" (dest)
```

変更されるもの

```
"memory"
```

オペランドについては制約条件を示した文字列とそれに対応したCの式で指定することになります。

まず出力オペランドの指定です。"=&S" (d0)で変数d0が"=&S"という制約を持っています。これは次のような意味になります。

- 「=」asmが終わったあとに変更された結果が指定された変数(d0)に代入される
- 「&」asmを実行する前に変更される
- 「S」%esiレジスタに割り当てる

これは最初のレジスタ制約なので、アセンブラテンプレートで%0と参照できます。また他のオペランドの"0"という制約としても使われます。

次が "=&D" (d1)で変数 d1 が "=&D" という制約を持っています。これも「=&」は同様で、「D」は%edi レジスタに割り当てるという意味になります。これは次のレジスタ制約なので、%1で参照できます。またオペランドでは "1" という制約になります。

次は "=&a" (d)は変数 d2 が "=&a" という制約を持ちます。「a」は %eax レジスタに割り当てることになります。%2 で参照でき、"2" という制約になります。

その次が入力オペランドの指定です。"0" (src)で変数 src に "0" という制約を付けます。"0" という制約は、すでに説明した通り、%esi レジスタに割り当てられているので、src の値を %esi レジスタに設定することになります。

"1" (dest)で変数 dest に "1" という制約を付けます。"1" という制約も、すでに説明した通り%edi レジスタに割り当てられているので、dest の値を %edi レジスタに設定することになります。

変更されるものとして "memory" を設定しておくことで、この asm 命令を実行するとメモリの内容が変更されてしまうということを GCC に伝えます。これにより最適化してメモリにある内容をレジスタに保持したままにしないようにします。その他に影響があるレジスタがあればそれらを記述しておくことができます。

以上をまとめると次のように解釈されます。

```
src 変数の値を %esi レジスタに設定します
  入力オペランドの "0" (src)と出力オペランドの "=&S" (d0)による
dest 変数の値を %edi レジスタに設定します
  入力オペランドの "1" (dest)に出力オペランドの "=&D" (d1)による
以下のアセンブラコードが埋め込まれる
   1:  lodsb
       stosb
       testb %al,%al
       jne 1b
%esi レジスタが変数 d0 に代入されます
  出力オペランドの "=&S" (d0)による
%edi レジスタが変数 d1 に代入されます
  出力オペランドの "=&D" (d1)による
%eax レジスタが変数 d2 に代入されます
  出力オペランドの "=&a" (d2)による
```

なお、__asm__ の後ろの __volatile__ はコンパイラの最適化を防ぐためのキーワードです。最適化によってコードが除去されてしまうことを防ぎます。

レジスタ制約

このようにGCCのasm命令では、オペランドに制約条件を指定することでどのレジスタを使うかを指定することができます。

"g" などを使えば適当なレジスタにGCCが割り当ててくれます。このように特定のレジス

タと指示しなくてすむ場合は、GCCは前後のCのコードでのレジスタの使用状況なども考慮して無駄なレジスタの割り当てをしないようにしてくれます。

x86では次の制約を使うことで特定のレジスタへの割り当てを行うことができます。

レジスタ	制約
eax	a
ebx	b
ecx	c
edx	d
edx:eax	A
edi	D
esi	S
fp	f
st(0)	t
st(1)	u
xmm SSE	x
MMX	y

このあたりはアーキテクチャによって異なります。これはGCCのソースのgcc/config/<アーキテクチャ>/<アーキテクチャ>.h(gcc/config/i386/i386.hなど)のREG_CLASS_FROM_LETTER(C)といったマクロなどで設定されています。Cが制約文字で、その文字に対応するレジスタをどれにすべきかをかえすマクロになっています。

詳しいことはgcc.infoのMachine Constraintsなどを参照してください。

まとめ

GCCのasm命令を使うとCの中でインラインアセンブラを使うことができます。その時にCの変数とアセンブラのレジスタとの対応をオペランドの制約文字列によって割り当てを制御できることを説明しました。

—— Fumitoshi Ukai

HACK #24 GCCのビルトイン関数による最適化を活用する

本Hackでは、GCCの最適化が効きやすいコードを書く方法について解説します。

GCCによる最適化の例

次のC言語のプログラムをgccでコンパイルすると、どのようなコードが得られるでしょ

うか？　特に、関数冒頭の2つのstrlen関数の呼び出しはどうコンパイルされるでしょうか？

```
//
//  User-Agent ヘッダを作成する
//
#define USER_AGENT_HDR_NAME "User-Agent: "
char *get_user_agent_hdr(const char *hdr_value) {
  assert(hdr_value != NULL);
  const size_t name_len  = strlen(USER_AGENT_HDR_NAME);
  const size_t value_len = strlen(hdr_value);
  char *hdr = malloc(sizeof(char) * (name_len + value_len + 1));
    // 整数オーバーフローのチェックは略
  if (hdr)
    sprintf(hdr, "%s%s", USER_AGENT_HDR_NAME, hdr_value);
  return hdr;
}
```

実際にプログラムを gcc -fverbose-asm -S でアセンブリ言語に変換してみたところ、次のような結果を得ることができました。

```
1: get_user_agent_hdr:
2:       pushl   %ebp    #
3:       movl    %esp, %ebp      #,
4:       subl    $24, %esp       #,
5:       movl    $12, -4(%ebp)   #, name_len
6:       subl    $12, %esp       #,
7:       pushl   8(%ebp) # hdr_value
8:       call    strlen  #
         ...
```

5行目に注目してください。なんと、1つ目のstrlen関数の呼び出しが、コンパイル時に即値($12)に変換されています。

ビルトイン関数

実はGCCは、strlenをはじめとする、C言語の規格で定められた主要な関数を、libcとは別に内部に持っています。これをGCCではビルトイン関数と呼んでいます。そして「strlen関数に文字列リテラルを渡した場合」など特定のケースでは、GCCはビルトイン関数を利用して出力するコードを最適化(高速化)してくれます。

最適化の対象になる関数

GCCがこのような最適化の対象にする関数は、非常に多岐にわたっています。一番身近なところでは、printf関数にさまざまな形で最適化が施されます。その他に以下のような関数

も最適化の対象です。

- 数学関係の関数(abs、acos、asin...)
- 文字種判定の関数(isalnum、isalpha、iscntrl...)
- 文字列操作の関数(strcat、strchr、strcmp...)

詳細は、GCCオンラインマニュアルの「Other built-in functions provided by GCC」という章にまとめられていますので、興味があれば参照すると良いでしょう。

なお、デバッグの都合などでこの種の最適化を行いたくない場合は下のように、-fno-builtinというオプションを使用してコンパイルしましょう。

```
% gcc -fno-builtin foo.c
```

ソースコード中でビルトイン関数を直接使う

GCCの最適化に身をゆだねるのではなく、「Other built-in functions provided by GCC」で紹介されているような__builtin_foobar()という関数をソースコード内で直接使用して、GCCに高速なコードを出力させることもできます。ここでは、ハッカーが好んで行う処理である「ビット数え」をとりあげてみます。

```
int __builtin_clz(unsigned int x);
```

という、「引数xの先頭何ビットが"0"であるかを返す関数」を試してみましょう。clzは、count-leading-zeroの略です。

```
const unsigned int x = 0x0fffffffU; // 最初の4bitは0、残り28bitは1
printf("%d\n", __builtin_clz(x));
```

このコードを、gcc -O2でコンパイルすると、見事、次のような最適化されたコードが出力されます。printfの第2引数を、GCCがコンパイル時に計算して4にしてくれています。

```
.LC0:
        .string "%d\n"
(略)
        pushl   $4
        pushl   $.LC0
        call    printf
```

なお、この手のHackについては「**[Hack #022] GCCのGNU拡張入門**」の解説も非常に参考になります。

文字列リテラルへのポインタに注意

最近のプログラミング作法として、「#define はなるべく使わないようにする」というものがあります。その作法に従い、最初のプログラムを次のように書き換えてみましょう。

```
// よくない例
static const char *USER_AGENT_HDR_NAME = "User-Agent: ";
char *get_user_agent_hdr(const char *hdr_value) {
  size_t name_len = strlen(USER_AGENT_HDR_NAME);
  ...
```

するとどうでしょう、意外なことに、GCCは1つ目のstrlen関数の呼び出しを即値に変換してくれなくなってしまいました。libc の strlen 関数が呼び出されてしまっています。

```
...
pushl   USER_AGENT_HDR_NAME
call    strlen
...
```

実は、文字列リテラルを指すポインタにもこの種の最適化を効かせたい場合、変数にconst修飾がもう1つ必要です。次のようにすれば、結果は一番最初のプログラムとほぼ同じになります。

```
// よい例
static const char* const USER_AGENT_HDR_NAME = "User-Agent: ";
//(別の)よい例
static const char USER_AGENT_HDR_NAME[] = "User-Agent: ";
```

「ポインタの参照しているメモリの内容」が一定であることだけでなく、「ポインタの参照先」も一定であると明言しなければ、GCCはstrlen関数の呼び出しを安心して即値にできないというわけです。

まとめ

変更する予定のない変数を宣言するときは、必ず const で修飾するようにしましょう。特に、文字列リテラルへのポインタを宣言するときは、2つめのconstも忘れずに付けるようにしましょう。GCCがビルトイン関数を使ってあなたのコードを最大限に高速化してくれます。また、ビルトイン関数をソースコード中で明示的に使用して、GCCに高速なコードを出力させることも可能です。

—— Yusuke Sato

HACK #25 glibcを使わないでHello Worldを書く

本Hackでは、システムコールだけを用いるプログラミングを紹介します。このHackを使うと、小さなプログラムを小さなバイナリとして出力することができます。

いわゆる"Hello World"は、C言語であれば5行くらいで書くことができます。しかし、そのプログラムをgccでコンパイル・リンクすると、動的リンクであっても5KBほどのバイナリが生成されてしまいます。本Hackでは、「どうして5行のプログラムが5KBになってしまうのか」「もっと小さなバイナリを出力するにはどうしたらいいか」を、誰にでも理解できる形で見ていきます。

5KBの内訳

早速、ナゾの5KBの内訳を見てみましょう。gccに-vオプションを付けて、Hello Worldプログラムのコンパイル・リンクを行ってみます。

```
% gcc -o hello -v hello.c
...
/usr/libexec/gcc/i386-redhat-linux/3.4.4/collect2 --eh-frame-hdr
  -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crt1.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crti.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/crtbegin.o
  -L/usr/lib/gcc/i386-redhat-linux/3.4.4 -L/usr/lib/gcc/i386-redhat-linux/3.4.4
  -L/usr/lib/gcc/i386-redhat-linux/3.4.4/../../.. /tmp/ccOPEg3W.o -lgcc
  --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
  /usr/lib/gcc/i386-redhat-linux/3.4.4/crtend.o
  /usr/lib/gcc/i386-redhat-linux/3.4.4/../../../crtn.o
% wc -c hello
4712 hello
```

すると、このような出力が得られました。どうやらgccコマンドは、libc.soをダイナミックリンクするだけではなく、何やらいろいろな.oファイルをhelloバイナリにリンクしていることがわかります。

これらのcrt*.oファイルには、簡単に言うとglibcの初期化を行うためのコードが含まれています。従って、Hello Worldプログラムを、まずglibc(libc.so)に含まれる関数を使用しないように記述し、その上で、crt*.oをリンクしないようにすれば、非常に小さなバイナリを得ることができるはずです。

小さなHello Worldを書く

Hello World程度のプログラムであれば、printfなどのlibcに含まれる関数を使うまでもなく、writeシステムコールを直接呼ぶことでも実現できます。ただし、次のように、直接

write関数やsyscall関数を使う方法ではうまくいきません。

```
#include <unistd.h>
...
  write(1, "Hello World\n", 12);
```

なぜならここでのwriteやsyscallは、「システムコールそのもの」というよりはglibc（libc.so）に含まれる単なる“関数”だからです。システムコールを呼ぶ方法、つまりカーネルモードに入る方法は、同じLinuxであってもアーキテクチャごとに異なるため、glibcがその差異を吸収してくれているわけです。これは大変ありがたいことですが、ここではglibcに頼らずに直接システムコールを呼ぶ方法を考え出さなければなりません。

システムコールの呼び方をカーネルのソースコードで把握する

実はx86の場合は、「**[Hack #59]　システムコールはどのように呼び出すか**」を読めば、システムコールを直接呼ぶ方法は、レジスタの使い方を含めすべてわかります。しかし、x86以外を利用しているなどの理由でシステムコールの呼び方を知らなかったとしても心配はいりません。Linuxカーネルのソースコードの一部をコピーアンドペーストすれば、とても簡単にシステムコールを（glibcを使わずに！）呼び出すことができます。

参考にするファイルは“linux-2.6.x/include/asm-<アーキテクチャ>/unistd.h”です。このファイルを開くと、_syscallNという名前（N=0,1,2..）のマクロがあるはずです。これらのマクロを、あなたのソースコード（hello.c）にコピーアンドペーストしてしまいましょう。ここでは、ひとまず_syscall1と_syscall3および、これらが依存しているマクロだけを持ってくればよいでしょう。

システムコールだけでHello Worldを書く

こうしてできあがったソースコードが次のものです。

```
#include <asm/unistd.h>  // __NR_xxx
static int errno;

// --- カーネルのソースコードからコピーアンドペーストした部分
#define _syscall1(type,name,type1,arg1) \
  (略)
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
  (略)
// コピーアンドペースト ここまで ---

_syscall1(int, exit, int, status);
_syscall3(int, write, int, fd, const void*, buf, unsigned long, count);

void hello() {
```

```
  write(1, "Hello World!\n", 13);
  exit(0);
}
```

ここで下の2行は、次に紹介するような「システムコールを直接呼び出すCの関数」に展開されます。

```
_syscall3(int, write, int, fd,  const void*, buf, unsigned long, count);
_syscall1(int, exit, int, status);
```

x86_64での例を見てみましょう。どうやら、syscallというCPUの命令を使えばシステムコールを呼び出せるようですね。

```
int write(int fd, const void *buf, unsigned long count) {
  long __res;
  __asm__ volatile ("syscall"
        : "=a"(__res)
        : "0"(__NR_write), "D"((long)fd), "S"((long)buf), "d"((long)count)
        : "r11","rcx","memory");
 if ((unsigned long)__res >= (unsigned long)-127) {
   errno = -(__res);
   __res = -1;
 }
 return (int)__res;
}
```

hello関数は、上記のように展開された関数だけを用いて記述しました。このhello.cを、次のようにコンパイルしておきましょう。

```
% gcc -Os -fno-builtin -fomit-frame-pointer -fno-ident -c hello.c
```

フレームポインタを省略したり、サイズ優先の最適化(-Os)をかけたりして、生成されるオブジェクトが小さくなるように工夫しています。

リンクする

リンクは、gccコマンドではなくldコマンドで行います。ldコマンドは、デフォルトで_startという関数をエントリポイント(実行を開始する関数)にすることが多いのですが、_startはcrt*.oに含まれている関数であり、いまは存在していません。そこで、hello.cのhello関数をエントリポイントに指定してしまうことにします。

```
% ld --entry=hello -o hello hello.o
```

静的リンクで942バイトのバイナリができました。objdumpコマンドでこれ(hello)を逆ア

センブル(-d)してみると、とてもシンプルな構成になっていることがわかると思います。

がんばってダイエットする

リンクしたバイナリをまず、strip -sします。そのあと、readelfコマンドでセクションの一覧を出力し、サイズがゼロであるなど、不要と判断できるセクションをstrip -Rで取り去ってしまうのがよいでしょう。

```
% strip -s hello
% readelf -S hello
Section Headers:
 [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
 [ 0]                   NULL            00000000 000000 000000 00      0   0  0
 [ 1] .text             PROGBITS        08048094 000094 00005b 00  AX  0   0  4
 [ 2] .rodata           PROGBITS        080480ef 0000ef 00000e 01 AMS  0   0  1
 [ 3] .data             PROGBITS        08049100 000100 000000 00  WA  0   0  4
 (略)
% strip -R .data hello
```

最終的にスタティックリンクで488バイトまでダイエットできました。

```
% wc -c hello
488 hello
% file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked,
stripped
% ldd hello
not a dynamic executable
```

こんなに小さなバイナリですが、もちろん問題なく実行できます。

```
% ./hello
Hello World!
```

コマンドライン引数を取得するには

先ほどのHello Worldプログラムでは、エントリポイントの関数の引数はvoidとしていましたが、通常のmain関数のように、コマンドライン引数や環境変数のリストが欲しい場合もあります。そのような場合は、glibcのソースコードのglibc-2.4/sysdeps/<アーキテクチャ>/elf/start.Sを参考にして、hello関数をargc、argvを引数にして呼び出すような、エントリポイントとなる関数をアセンブラで記述すればOKです。

start.Sの_start関数は、argcやargvを含むいくつかの変数を引数にして、__libc_start_mainというCの関数を呼び出しています。ですから、これを簡略化し「argcとargvだけを引数にhelloを呼び出す」ようにすればよいでしょう。若干アセンブリ言語の知識が必要になってし

まうのが難点ですが、困難な作業ではありません。作業にあたっては、「**[Hack #58] プログラムが main()にたどりつくまで**」も参考になるでしょう。

まとめ

本Hackでは、システムコールだけを用いるプログラミングを紹介しました。このHackを使うと、ごく小さなプログラムをごく小さなバイナリとして出力することができます。組み込み用途などでは、このHackが実際に役立つこともあります(http://www.selinux.gr.jp/LIDS-JP/document/general/web_lids_busybox/main.html)。

紹介したのはLinuxやglibcのソースコードを参考にするという方法ですから、どんなアーキテクチャであっても、この Hack を参考にすれば極小のバイナリを出力することができます。極端な話、そのアーキテクチャのアセンブリ言語をよく理解していなくても大丈夫かもしれません。

—— Yusuke Sato

HACK #26 TLS(スレッドローカルストレージ)を使う

gccでは、__threadキーワードを使用することでTLSを使用することができます。

TLS(Thread-Local Storage)とは、どのスレッドも同じ名前の変数を使いながらも、実際に格納される値はスレッドごとに独立して保持できる領域のことを指します。

gccでは、__threadキーワードを使用することでTLSを使用することができます。例として、スレッドを3つ生成し、各スレッドを3つ生成し、各スレッドからグローバル変数とTLS変数を参照するというコードを次に示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREADS 3

__thread int tls;
int global;

void *func(void *arg)
{
  int num = (int) arg;
  tls = num;
  global = num;
  sleep(1);
  printf("Thread=%d tls=%d global=%d\n", num, tls, global);
}
```

```
int main()
{
  int ret;
  pthread_t thread[THREADS];
  int num;

  for (num = 0; num < THREADS; num++) {
    ret = pthread_create(&thread[num], NULL, &func, (void *) num);
    if (ret) {
      printf("error pthread_create\n");
      exit(1);
    }
  }
  for (num = 0; num < THREADS; num++) {
    ret = pthread_join(thread[num], NULL);
    if (ret) {
      printf("error pthread_join\n");
      exit(1);
    }
  }
  exit(0);
}
```

このプログラムを実行すると、以下の結果が得られます。

```
% ./threadlocal
Thread=0 tls=0 global=2
Thread=2 tls=2 global=2
Thread=1 tls=1 global=2
```

上記プログラムで、グローバル宣言した変数globalはスレッド間で共有していますので、最後に代入された値（ここでは2）を保持しています。これに対し、TLS変数tlsはグローバル変数と同じような変数名を使っていますが、実際はスレッドごとに異なる値を持つことができます。

TLSが使われる場面

__threadキーワードは、例えばglibcでは、エラー値errnoに対して使用されています。スレッドごとに異なるエラー値を保持できるようにするためです。

__threadキーワード自体は、gcc固有ですが、ポータブルに使用する方法としては、pthreadのスレッド固有データ（TSD: Thread-Specific Data）が用意されています。pthread_key_t型とpthread_key_create関数などがその例です。

TLSの実装

TLSは、例えばLinuxの場合は、Linuxカーネル2.6 + gcc 3.3 + glibc 2.3 + NPTL以降で活用することができます。というのも、TLSを実現するにはカーネル、glibc、gccなどさまざまな部分の対応が必要になります。

また、アーキテクチャによっても実装方法が大きく異なります。例えば、x86ではTLSを実現するためのスレッドレジスタとして%gsセグメントレジスタを利用しています。なかには、mipsのように余分なスレッドレジスタが残っていないため、なかなかTLSが実現できていないアーキテクチャもあります。

詳しい実装についての資料としては、「ELF Handling For Thread-Local Storage」(http://people.redhat.com/drepper/tls.pdf)、「The Native POSIX Thread Library for Linux」(http://people.redhat.com/drepper/nptl-design.pdf)などが良い手がかりとなるでしょう。

まとめ

本HackではGNU/Linuxの環境でTLS(スレッドローカルストレージ)を使う方法を紹介しました。

—— Masanori Goto

HACK #27 glibcでロードするライブラリをシステムに応じて切り替える

Linuxで動作するglibcの動的ローダにはプラットホームやHWCAPごとに、ライブラリを切り替えることができます。

Linuxで動作するglibcの動的ローダは、x86やsparcといったアーキテクチャの中で、さらに細かくプラットホーム(i386、i486、i586、i686)やHWCAP(HardWare CAPabilities、x86ではMMXやSSEなど)ごとに、ライブラリを切り替えるという隠れ技を持っています。

例えば、「**[Hack #62] dlopenで実行時に動的リンクする**」で紹介されているdlopen(3)のプログラムdlsayについて存在しないライブラリを対象に実行させたときにstraceをかけた時の様子を次に示します。

```
% strace -f ./dlsay non-existed symbol
...
open("/lib/tls/i686/sse2/cmov/non-existed", O_RDONLY) = -1 ENOENT (No such file or
directory)
stat64("/lib/tls/i686/sse2/cmov", 0xbfe29570) = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/sse2/non-existed", O_RDONLY) = -1 ENOENT (No such file or
directory)
```

```
stat64("/lib/tls/i686/sse2", 0xbfe29570) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/non-existed", O_RDONLY) = -1 ENOENT (No such file or
directory)
...
open("/lib/sse2/non-existed", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/sse2", 0xbfe29570)         = -1 ENOENT (No such file or directory)
open("/lib/cmov/non-existed", O_RDONLY) = -1 ENOENT (No such file or directory)
stat64("/lib/cmov", 0xbfe29570)         = -1 ENOENT (No such file or directory)
open("/lib/non-existed", O_RDONLY)      = -1 ENOENT (No such file or directory)
...
```

途中で多数のルックアップをかけていることがわかります。この時よく注意してみると、"tls" や "i686"、"sse2"、"cmov" といったキーワードが見えます。これらは、通常のライブラリディレクトリを検索する前に、チェックする特別なディレクトリです。それぞれ少しずつ意味が異なります。

tls
: ディストリビューションがカーネル2.4と2.6両方に対応している場合、スレッドローカルストレージ（TLS）に対応した場合にチェックされます。例えば、カーネル2.6+TLS を使っているシステムで、LinuxThreads もインストールしているシステムでは、LD_ASSUME_KERNELという変数に値を設定するかどうかでTLSを使うかの制御が可能です（なお、最近のglibcではLinuxThreadsのサポートが行われなくなりつつあります）。

i686（使っているプロセッサクラス（プラットホーム）ごとに変化する名前）
: この場合i686クラスプロセッサの上で動作させているので、チェックされています。古い i586 マシンの場合、i686 ではなく i586 ディレクトリがチェックされます。

sse2、cmov
: sse2やcmov（Conditional MOVe）といった、アーキテクチャの中でもサポートしているプロセッサとサポートしていないプロセッサがある場合に使用されます。これをHWCAP といいます。

プラットホームと HWCAP 情報が渡される仕組み

上記のプラットホーム名やHWCAPは、実はカーネルからプログラムに制御が移るときに、ELFの情報の一部として一緒にAUXV（AUXiliary Vector）というデータとして一緒に渡されてきます。この情報を表示する方法は、プログラム起動時にLD_SHOW_AUXVという環境変数を与えます。すると、glibc の動的ローダが各種情報を出力してくれます。

```
% LD_SHOW_AUXV=1 /bin/echo
AT_SYSINFO:      0xffffe400
AT_SYSINFO_EHDR: 0xffffe000
AT_HWCAP:    fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x8048034
AT_PHENT:        32
AT_PHNUM:        7
AT_BASE:         0xb7fd0000
AT_FLAGS:        0x0
AT_ENTRY:        0x8048a90
AT_UID:          4107
AT_EUID:         4107
AT_GID:          400
AT_EGID:         400
AT_SECURE:       0
AT_PLATFORM:     i686
```

この中にあるAT_PLATFORMがプラットホーム、AT_HWCAPがHWCAPを指します。これらの情報はカーネルが起動時にCPUから読み出して保持しています。HWCAPについては、x86系プロセッサの場合、CPU フラグを読み出すことで実現されています。

プラットホームや HWCAP に応じて使い分ける

さて、あるライブラリを i386 用と、i686+SSE2 用の 2 つ作成したとしましょう。そして、dlopen()でロードするライブラリを、SSE2を持っているマシンとそうでないマシンとで切り替えたいとしましょう。

通常推奨される方法は、ライブラリ中の自前コードで、SSEやi686をサポートしているかどうかをチェックするというものです。ただ、いちいちそういったコードを組み込むのが面倒な場合(例えばコンパイラレベルの最適化で切り分けたいときなど)、前述のプラットホーム、HWCAP 情報が役立ちます。

使い方は簡単です。dlopen やライブラリを置くディレクトリ(/lib、/usr/lib、LD_LIBRARY_PATH など)に通常の汎用ライブラリを置きます。例えばここでは i386 用のライブラリを置きます。そして、そのディレクトリの下にi686/sse2というディレクトリを作成し、i686+SSE2で最適化したライブラリを置きます。

これだけで、glibcは勝手にi686+SSE2最適化ライブラリを優先してロードするようになります。

LD_HWCAP_MASK 変数を設定する

前述のHWCAPですが、実はすべてのフラグを総当たりでチェックすると組み合わせ爆発

を起こしてしまうため、glibcでは、ロードするパス名に組み込む名前要素を、内部で限定してチェック対象にしています。特に、最新の glibc 2.3.9x シリーズでは、sse2 だけがチェック対象になっています。

しかし、例えばi386用、i686+MMX用、i686+SSE2用の3種類をチェックしたいと思うかも知れません。それらのフラグをチェック対象にする場合、LD_HWCAP_MASK環境変数によって制御することができます。例として、MMXとSSE2をチェック対象にした場合を示します。

```
LD_HWCAP_MASK=0x04800000
```

このマスク位置は、プロセッサの種類に依存する値です。Pentium 4 では、ビット 23 が MMX、ビット 26 が SSE にあたります。

なお、自分でdlopen()するようなプログラムの場合は良いのですが、ld.soによって動的ライブラリをロードしてしまう場合、glibcはLD_HWCAP_MASKの設定を見ずに/etc/ld.so.cacheにキャッシュされたパス名のライブラリをロードしてしまいます。その場合、自前で最適化したライブラリは読んでくれないことに注意が必要です。

まとめ

本Hackでは、プラットホームやHWCAPを利用した、ライブラリ切り替え方式について説明しました。なお、ここで説明した方法は、システムのために用意されている隠れ技です。使う場合には互換性などに関して注意が必要です。

プラットホームやHWCAPを活用しているディストリビューションの1つがDebian GNU/Linux sargeです。通常のglibcではチェックされない、cmovというHWCAPマスクをデフォルトで追加することにより、VIA C3のようにcmovという命令を持っていないにもかかわらず、プラットホームがi686というような通常と異なるCPUで安全にバイナリを実行できる仕組みを入れています。

—— Masanori Goto

HACK #28 リンクされているライブラリによってプログラムの動作を変える

GNU拡張を用いてweakシンボルを利用する方法を紹介します。

weakシンボルを用いると、リンクされているライブラリによってプログラムの動作を変えることができます。本HackではGNU拡張を用いてweakシンボルを利用する方法を紹介します。

それではさっそくサンプルコードを見てみましょう。このプログラムでは、libm.soに含ま

れるsqrt()関数があるときは利用し、ない場合はその旨のメッセージを表示します。

weak.c

```
#include <stdio.h>
extern double sqrt(double x) __attribute__ ((weak));

void func() {
    if (sqrt) {
        printf("%f\n", sqrt(10.0));
    } else {
        printf("sqrt isn't available\n");
    }
}

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

実行結果は次の通りです。

```
% gcc -fPIC -shared -o weak.so weak.c  # weak.so を作成する
% gcc weak-main.c ./weak.so -lm; ./a.out
3.162278
% gcc weak-main.c ./weak.so; ./a.out
sqrt isn't available
```

このプログラムのポイントはweak.cの2行目の部分です。__attribute__ ((weak))というGNUの拡張を使ってweakシンボルを宣言しています。

weak.soに含まれるシンボルをnmで見ると、sqrtはweakシンボルになっています。

```
% nm a.out |grep sqrt
         w sqrt
```

weakシンボルは、シンボルの定義(関数の実体など)がリンク時に見つからなかった場合、0に初期化されます。よって上のプログラムでは、sqrt関数がリンク時に見つからないときはsqrt = 0となり、if (sqrt)のelseのブロックが実行されます。

weakシンボルのメリット

上の例と似たようなことは、autoconfなどで行われている#ifdef HAVE_SQRTのようなマクロでもできます。

```
#ifdef HAVE_SQRT
        printf("%f\n", sqrt(10.0));
#elif
        printf("sqrt isn't available\n");
#endif
```

しかし、この場合、weak.soの動作はコンパイル時に固定されるという違いがあります。このため、weak.soがHAVE_SQRT = 1でコンパイルされた場合、weak.soを利用するプログラム(クライアント)はsqrt()の呼び出しのために、必ずlibm.soをリンクする必要が生じます。

libm.soのような基本的なライブラリの場合は問題はほとんどないと思いますが、特殊なライブラリの場合(一般的でないライブラリやlibpthreadのようにプログラムの性能に影響が生じるライブラリなど)、クライアントはそのライブラリをリンクするか選択したい場合もあります。weakシンボルの手法が役立つのはこのような場合です。

まとめ

本Hackではweakシンボルを用いて、リンクされているライブラリによってプログラムの動作を変える方法を紹介しました。GNU拡張を用いるため移植性が下がってしまうのが難点ですが、ライブラリの依存関係の柔軟性を高めるのに役立つと思います。

—— Satoru Takabayashi

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__((visibility("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__((visibility("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__((visibility("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 #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

バージョンスクリプトを使うことで利用できる、バージョンドシンボルについて説明します。

「**[Hack #29]　ライブラリの外に公開するシンボルを制限する**」で紹介しているように、バージョンスクリプトを使うことでシンボルの公開、非公開を制御することができます。ここでは、バージョンスクリプトを使うことで利用できる、バージョンドシンボルについて説明します。

サンプルの導入

まず、サンプルとして2つの引数をとってどちらかの最大値を返す関数maxと、その関数を含むライブラリlibmaxを作成します。

libmax1.c

```
#include <stdio.h>
int max(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
```

libmax1.h

```
extern int max(int a, int b);
```

vertest1.c

```
#include <stdio.h>
#include "libmax1.h"
int main(void)
{
  printf("max(1, 2)=%d\n", max(1, 2));
  return 0;
}
```

実行例

```
% gcc -fPIC -c -o libmax1.o libmax1.c
% gcc -shared -Wl,-soname,libmax.so.1 -o libmax.so.1.0 libmax1.o
% ln -s libmax.so.1.0 libmax.so.1
% gcc -L. -lmax -o vertest1 vertest1.c
% LD_LIBRARY_PATH=. ./vertest1
max__1
max(1, 2)=2
```

上記では、libmax.so.1という共有ライブラリファイル名を持つライブラリlibmax.so.1.0を作成し、それを呼び出すプログラムvertest1を作成、実行しています。vertest1は、動的ローダによって実行時にlibmax.so.1がリンクされ、関数maxの結果が返されています。

バージョンドシンボル

ここまで、例として取り上げた2つ引数をとるmaxを、3つの引数をとるように変更したいと考えたとします。ここで、バージョンドシンボルを使うことで、maxというシンボル名はそのままで、別々の動作をする複数のバージョンを持たせることができます。さらに、古いプログラムは2引数版maxを、新しいプログラムは3引数版maxを別々に呼ばせるよう制御できるのです。以下の例を見てみましょう。

libmax2.c

```
#include <stdio.h>
int max__1(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
int max__2(int a, int b, int c)
{
  int d = a > b ? a : b;
  printf("max__2\n");
  return (d > c ? d : c);
}
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

libmax2.h

```
extern int max(int a, int b, int c);
```

vertest2:

```
#include <stdio.h>
#include "libmax2.h"
int main(void)
{
  printf("max(1, 2, 3)=%d\n", max(1, 2, 3));
  return 0;
}
```

新しいプログラムvertest2では、3引数版maxを使っています。そのため、libmax2.hのmaxの引数も3つとして宣言されています。しかし、このままlibmax.so.1を3引数版に変更してしまうと、過去に生成したvertest1も3引数版で動作してしまいます。そこで、新しく作成する共有ライブラリlibmax2.cでは、2引数版と3引数版の両方を、maxとは別の関数名を付けて書きます。さらに、.symverアセンブラ命令で、同じシンボル名maxに対して@の後にバージョン名を付けて、maxからそれぞれエイリアスします。ここでは、古い2引数版をLIBMAX_1.0、新しい3引数版をLIBMAX_2.0としましょう。

コンパイルするときは、次のバージョンスクリプトをリンカに渡し、シンボルにバージョンを設定します。このファイルでは、LIBMAX_1.0とLIBMAX_2.0を定義し、それぞれmaxというシンボル名を各バージョン別に持っていることを表しています。また、LIBMAX_2.0がLIBMAX_1.0の次バージョンである(依存しているとも言う)ことも表しています。

libmax2.def

```
LIBMAX_1.0 {
  global: max;
  local: *;
};
LIBMAX_2.0 {
  global: max;
} LIBMAX_1.0;
```

実際にコンパイルして実行させてみましょう。

```
% gcc -fPIC -c -o libmax2.o libmax2.c
% gcc -shared -Wl,-soname,libmax.so.1 -Wl,--version-script,libmax2.def -o
  libmax.so.1.0 libmax2.o
% gcc -L. -lmax -o vertest2 vertest2.c
% LD_LIBRARY_PATH=. ./vertest2
max_2
max(1, 2, 3)=3
% LD_LIBRARY_PATH=. ./vertest1
max_1
max(1, 2)=2
```

興味深い点は、古いバイナリvertest1は、古い2引数版maxを呼び出していることです。それぞれのバイナリに対してreadelfをかけてみると、何が起きているか分かります。

```
% readelf -a vertest1 | grep max
 0x00000001 (NEEDED)                     Shared library: [libmax.so.1]
08049700  00000307 R_386_JUMP_SLOT   00000000   max
     3: 00000000    37 FUNC    GLOBAL DEFAULT  UND max
    64: 00000000    37 FUNC    GLOBAL DEFAULT  UND max
% readelf -a vertest2 | grep max
 0x00000001 (NEEDED)                     Shared library: [libmax.so.1]
0804968c  00000307 R_386_JUMP_SLOT   00000000   max
     3: 00000000    69 FUNC    GLOBAL DEFAULT  UND max@LIBMAX_2.0 (3)
    76: 00000000    69 FUNC    GLOBAL DEFAULT  UND max@@LIBMAX_2.0
  000000: Version: 1  File: libmax.so.1  Cnt: 1
% readelf -a libmax.so.1
...
Symbol table '.symtab' contains 67 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    50: 000004f1    69 FUNC    LOCAL  DEFAULT   11 max__2
    52: 000004cc    37 FUNC    LOCAL  DEFAULT   11 max__1
    63: 000004f1    69 FUNC    GLOBAL DEFAULT   11 max@@LIBMAX_2.0
    65: 000004cc    37 FUNC    GLOBAL DEFAULT   11 max@LIBMAX_1.0
...
Version definition section '.gnu.version_d' contains 3 entries:
  Addr: 0x00000000000002e8  Offset: 0x0002e8  Link: 3 (.dynstr)
  000000: Rev: 1  Flags: BASE  Index: 1  Cnt: 1  Name: libmax.so.1
  0x001c: Rev: 1  Flags: none  Index: 2  Cnt: 1  Name: LIBMAX_1.0
  0x0038: Rev: 1  Flags: none  Index: 3  Cnt: 2  Name: LIBMAX_2.0
  0x0054: Parent 1: LIBMAX_1.0
```

ご覧のように、古いバイナリvertest1には、バージョンが付いていない関数maxを呼び出そうとして、max__1(max@LIBMAX_1.0)が呼び出されています。新しいバイナリvertest2には、maxというシンボルにバージョンが付いていて、関数LIBMAX_2.0のmaxが呼び出すように指定されています。

こうして、古いバイナリでは古いバージョンを、新しいバイナリでは新しいバージョンを、同じシンボル名を使って使い分けることができるようになりました。

3種類あるバージョンの表現方法

ここからは、先ほどのバージョンがどのように適用されるのか、maxに4種類の異なるバージョン名がついている例で考えてみましょう。

libmax3.def

```
LIBMAX_1.0 {
  global: max;
  local: *;
};
LIBMAX_1.5 {
  global: max;
} LIBMAX_1.0;
LIBMAX_2.0 {
  global: max;
} LIBMAX_1.5;
```

libmax3.c

```
#include <stdio.h>
int max__0(int a, int b)
{
  printf("max__0\n");
  return (a > b ? a : b);
}
int max__1(int a, int b)
{
  printf("max__1\n");
  return (a > b ? a : b);
}
int max__1_5(int a, int b)
{
  printf("max__1_5\n");
  return (a > b ? a : b);
}
int max__2(int a, int b, int c)
{
  int d = a > b ? a : b;
  printf("max__2\n");
  return (d > c ? d : c);
}
__asm__ (".symver max__0,max@");
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__1_5,max@LIBMAX_1.5");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

このファイルを先ほど同様にコンパイルし、libmax.so.1 に対して readelf を行った結果が以下となります。

```
% readelf -a libmax.so.1
Symbol table '.symtab' contains 74 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    51: 0000065b    85 FUNC    LOCAL  DEFAULT   11 max__2
```

```
    53: 000005f1    53 FUNC    LOCAL  DEFAULT   11 max__1
    54: 00000626    53 FUNC    LOCAL  DEFAULT   11 max__1_5
    57: 000005bc    53 FUNC    LOCAL  DEFAULT   11 max__0
    63: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_1.0
    64: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_1.5
    65: 00000000     0 OBJECT  GLOBAL DEFAULT  ABS LIBMAX_2.0
    66: 00000626    53 FUNC    GLOBAL DEFAULT   11 max@LIBMAX_1.5
    68: 000005bc    53 FUNC    GLOBAL DEFAULT   11 max@
    70: 0000065b    85 FUNC    GLOBAL DEFAULT   11 max@@LIBMAX_2.0
    72: 000005f1    53 FUNC    GLOBAL DEFAULT   11 max@LIBMAX_1.0
...
Version definition section '.gnu.version_d' contains 4 entries:
  Addr: 0x000000000000036c  Offset: 0x00036c  Link: 3 (.dynstr)
  000000: Rev: 1  Flags: BASE  Index: 1  Cnt: 1  Name: libmax.so
  0x001c: Rev: 1  Flags: none  Index: 2  Cnt: 1  Name: LIBMAX_1.0
  0x0038: Rev: 1  Flags: none  Index: 3  Cnt: 2  Name: LIBMAX_1.5
  0x0054: Parent 1: LIBMAX_1.0
  0x005c: Rev: 1  Flags: none  Index: 4  Cnt: 2  Name: LIBMAX_2.0
  0x0078: Parent 1: LIBMAX_1.5
```

バージョン名として、@の後にバージョンがないもの、@の後にバージョンがあるもの、@@の後にバージョンがあるもの、全部で3種類の異なる表記方法があることに気が付くはずです。それぞれについて、以下で説明します。

@ の後にバージョンがないもの(max@)

このときのシンボルを、バージョン未指定のベースシンボルと呼びます。上記.gnu.version_dエントリ中ではBASEとなりIndex値は1になっています。バージョン未指定のベースシンボルが使われるのは、例えばvertest1を実行する際、libmax.so.1の中にmax__0とmax__2のみが存在するライブラリの場合に古いmax__0を使用するようにするためです。ただし、@の後にバージョンがついているエントリが存在する場合は、このベースシンボルは使用されません。

libmax3.c が以下の .symver のみ含むとき

```
__asm__ (".symver max__0,max@");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

ベースシンボルが使われる

```
% LD_LIBRARY_PATH=. ./vertest1
max__0
max(1, 2)=2
```

@ の後にバージョンがあるもの（max@LIBMAX_1.0、max@LIBMAX_1.5）

シンボル名maxに対し、@の後に付いたバージョン名が異なる複数のシンボルを設定することができます（これはGNU拡張です）。.gnu.version_dエントリ中のIndex値は2以降になります。libmax3.defの例では、LIBMAX_1.5はLIBMAX_1.0に依存していますので、LIBMAX_1.0よりもIndex値はより大きい値になります。

バージョン未指定のベースシンボルが定義されているかどうかに関わらず、バージョン付きシンボルが複数存在する場合、古いバイナリvertest1が選択する関数maxはバージョンを最初に設定したmax@LIBMAX_1.0になります。

libmax3.cが以下のsymverを含むとき、

```
__asm__ (".symver max__0,max@");
__asm__ (".symver max__1,max@LIBMAX_1.0");
__asm__ (".symver max__1_5,max@LIBMAX_1.5");
__asm__ (".symver max__2,max@@LIBMAX_2.0");
```

バージョン付きシンボルの最初のバージョンが使用されます。

```
% LD_LIBRARY_PATH=. ./vertest1
max__1
max(1, 2)=2
```

@@ の後にバージョンがあるもの（max@@LIBMAX_2.0）

@@の後にバージョンがあるものをデフォルトバージョンと呼びます。デフォルトバージョンは、バージョンドシンボルを定義する際に必ず1つのみ存在しなければなりません。あるシンボルに対し、非デフォルトバージョンが存在するのにデフォルトバージョンは存在しなかったり、複数のデフォルトバージョンが存在していると、リンク時にエラーとなります。

デフォルトバージョンは、通常は最新シンボルバージョンを表すことになります。バージョンを指定してコンパイルしない限り、バイナリからリンクされるバージョンはいつでもデフォルトバージョンであるmax@@LIBMAX_2.0が選択されます。これまでの例で./vertest2をコンパイルするとmax__2が自動的に使用されていたのはそのためです。

バージョンつきライブラリを使ってコンパイルしたバイナリは、シンボルにデフォルトバージョンをつけて記録します。そのため、たとえライブラリがアップグレードしてLIBMAX_3.0を新たに定義し、LIBMAX_2.0に依存させた場合でも、vertest2はmax@LIBMAX_2.0をこれまで通り呼び出すことができます。

バージョンドシンボルのメリット

バージョンドシンボルを使用する最大のメリットはライブラリの拡張が容易になることです。前述のmaxという関数名を変更したいとき、これがアプリケーション内部の関数ならば、2引数版maxの他に3引数版max_arg3を追加するか、max自体の仕様を変更にしてしまうでしょう。しかし、ライブラリで公開している関数ならば、ライブラリのメジャーバージョンを変更する、新しい仕様の関数は別の名前(例えばnew_maxなど)に変更するといった、互換性対策を施す必要に迫られます。

しかし、特にlibmaxが広く使われているような場合、関数maxの仕様変更をするだけでメジャーバージョンの変更や別の関数名にすることは困難です。例えばlibcやlibpngなどを考えてみましょう。メジャーバージョンを変えてしまうと、これまでlibmax.so.1という名前で参照していたバイナリは、すべて動的リンクできずに動かなくなってしまいます。また、libcなどでは公開されている関数名がある程度決まってしまっています。そういった難しい問題をうまく解決してくれるのが、バージョンドシンボルなのです。

まとめ

ここでは、バージョンドシンボルについて例を交えて説明しました。ライブラリをメンテナンスしていくと、どうしても仕様変更したい局面にたびたび遭遇します。バージョンドシンボルを利用すると、バイナリの互換性を保ちながらライブラリ内部仕様が変更可能になります。

—— Masanori Goto

HACK #31 main()の前に関数を呼ぶ

main()の前に関数を暗黙的に呼ぶ方法を、GCCの拡張を使う方法と、C++のコンストラクタを使う方法の2通りで紹介します。

C/C++のプログラムで、main()の前に関数を暗黙的に呼びたいときがあります。ここではGCCの拡張を使った方法と、C++のコンストラクタを使った方法を紹介します。

やり方

GCCではmain()の前に呼ばれる関数を__attribute__((constructor))という拡張機能を使って定義できます。たとえば、次のプログラムではmain()の前にfoo()が呼び出されます。

```
#include <stdio.h>
__attribute__((constructor))
void foo() {
```

```
    printf("hello, before main\n");
}

int main () {
    printf("hello, world\n");
    return 0;
}
```

実行結果は以下の通りです。

```
% ./a.out
hello, before main
hello, world
```

__attribute__((constructor))はGCCの拡張機能であるため移植性がなくなります。GCCのその他の関数属性についてはGCCマニュアルと「**[Hack #22]　GCCのGNU拡張入門**」を参照してください。

一方、C++ではクラスのコンストラクタを使って、同様のことを移植可能な方法で行うことができます。次のプログラムでは無名の名前空間の中でfoo_callerというクラスのコンストラクタでfoo()を呼ぶようにしています。foo_callerクラスのオブジェクトcallerが作られるとfoo_callerのコンストラクタが呼ばれてfoo()が呼ばれる仕組みです。

```
#include <stdio.h>

void foo() {
    printf("hello, before main\n");
}
namespace { struct foo_caller { foo_caller() { foo(); } } caller; }

int main () {
    printf("hello, world\n");
    return 0;
}
```

ライブラリの場合

__attribute__((constructor))を使う方法でも、C++のコンストラクタを使う方法でも、どちらの場合も、

```
__attribute__((constructor))
void foo() {
    printf("hello, before main\n");
}
```

および、

```
void foo() {
    printf("hello, before main\n");
}
namespace { struct foo_caller { foo_caller() { foo(); } } caller; }
```

の部分を別ファイル(foo.cppなど)にして、arで静的ライブラリを作った場合は、foo()がmainの前に呼び出されなくなります。

```
% g++ -c foo.cpp
% g++ -c main.cpp
% ar r libfoo.a foo.o
% g++ main.o libfoo.a
% ./a.out
hello, world
```

libfoo.aを作らずにfoo.oをリンクした場合と、libfoo.soを作ってリンクした場合は問題ありません。

```
# foo.o をリンク
% g++ main.o foo.o
% ./a.out
hello, before main
hello, world

# libfoo.so を作ってリンク
% g++ -fPIC -shared -o libfoo.so foo.o
% g++ main.o ./libfoo.so
% ./a.out
hello, before main
hello, world
```

libfoo.aの場合でも、-Wl,--whole-archive libfoo.a -Wl,--no-whole-archiveというオプションを渡して、強制的にlibfoo.aに含まれるすべてのオブジェクトファイルをリンクするようにすれば、foo()が呼ばれるようになります。

```
% g++ main.o -Wl,--whole-archive libfoo.a -Wl,--no-whole-archive
% ./a.out
hello, before main
hello, world
```

ライブラリ内の静的オブジェクトに関する挙動は、使っているOSの種類やGCC、glibc、Binutilsのバージョンなどによって変わってくると思います。移植性を重視する場合は要注意です。

main()の後で関数を呼ぶ

静的オブジェクトのデストラクタを応用すると、main()の実行が終わった後で任意の関数を呼ぶことができます。__attribute__((constructor))と対をなす__attribute__((destructor))を使って定義することもできます。また、プログラム終了時に実行する関数はANSI Cのatexit()関数でセットすることもできます。

```
#include <stdio.h>

void foo() {
    printf("hello, after main\n");
}
namespace { struct foo_caller { ~foo_caller() { foo(); } } caller; }

int main () {
    printf("hello, world\n");
    return 0;
}

% ./a.out
hello, world
hello, after main
```

まとめ

通常、main()の前の暗黙的な関数呼び出しが必要になることはないと思いますが、LD_PRELOADを用いたHackをするといった場面では役に立つかもしれません。「**[Hack #63] Cでバックトレースを表示する**」で紹介するlibSegFault.soでは__attribute__((constructor))を使ってシグナルハンドラをセットしています。

—— Satoru Takabayashi

HACK #32 GCC が生成したコードによる実行時コード生成

GCCが生成したコードが実行中にコードを生成する場合と、スタックオーバーフローといったセキュリティ上の問題との関連を解説します。

まず、GCCが生成したコードが生成するコードについて説明します。

トランポリン

GCCによって生成されたコード自身が、実行中にコードを生成することがあります。ただしこれは、C言語の関数内で別の関数が定義されていて、その内側の関数のアドレスがさらに別の関数に渡され、渡されたアドレス経由で内側の関数が呼び出され、そこで外側の関数

で定義されたローカル変数がアクセスされる、というごくごく稀な場合にのみ起こることです。

具体的にコードを見ていきます。

```
void other(void (*funcp)()) {
  funcp();
}

void outer(void) {
  int a = 10;

  void inner(void) {
    printf("outer's a is %d\n", a);
  }

  other(inner);
}
```

ここでouter()を呼び出した場合を考えます。outer()が、関数inner()のアドレスを引数として関数other()を呼び出しています。呼び出されたother()は渡されたアドレスを経由してinner()を呼び出します。inner()は、外側のouter()関数のローカル変数aの値を取得しています。

inner()まで呼び出された際のスタックの様子は、図3-1の通りです。inner()が変数aの値を得るためには、関数フレームをいくつか飛び越えてouter()のフレーム内にアクセスすることになります。そのためには、inner()は変数aの位置を把握しなければなりません。

ここでもし、inner()実行中のスタックポインタやベースポインタから変数aまでの距離が一定だったならば、GCCは生成するコードに単にその距離を埋め込んでしまえば済みます。しかし悪いことに、この距離は実行してみないことには判りません。さらに言うと、inner()の呼び出しごとに変わるかもしれません。

そこで、outer()はスタック上にコードを生成します。outer()が生成したコードとinner()が協力することで、inner()からouter()のローカル変数へのアクセスが可能となります。

outer()はスタック上に次のコードを生成します。

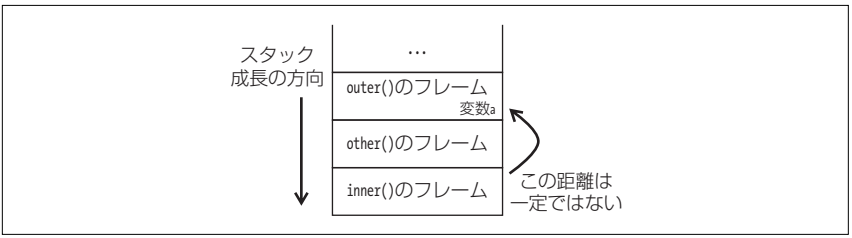


図3-1 スタックの状態

```
outer()のローカル変数のアドレスをあるレジスタに格納する
inner()の先頭アドレスにジャンプする
```

そして、inner()のアドレスとして、本当のアドレスの代わりに、生成したこのコードのアドレスを other()に渡します。other()は、関数ポインタ funcp を通して inner()を呼び出したつもりが、実はこのコードが実行されることになります。すると、あるレジスタにouter()のローカル変数のアドレスが入った状態で inner()が呼び出され、inner()はそれを手がかりに outer()のローカル変数にアクセスできるという仕掛けです。

ここでスタック上に生成されたコードは「トランポリン」と呼ばれます。呼ばれると、ちょっとした仕事をしてすぐに制御を inner()に移すため、そう呼ばれるのでしょう。

GCC 拡張

C 言語の ISO 規格、つまり JIS 規格では、関数内で別の関数を定義することは許されていません。この機能は GCC 独自の機能です。

セキュリティとの関係

スタック上へのコード生成は、セキュリティ上の問題をはらんでいます。バッファオーバーフローという攻撃が、しばしば、スタック上に不正に書き込んだ命令を実行するという方法で行われるためです。このため、安全のためにはスタック上の命令は実行できないようにしておくにこしたことはありません。このため、スタックやヒープ上に置かれた命令の実行は OS が禁止していることが一般的です。実行しようとするとプロセッサが例外を発します。これにより、UNIX 系 OS ではシグナルが発生します。

しかし、それではスタック上のトランポリンを実行できません。そこでGCCは、スタック上に置かれたコードの実行を許したいオブジェクトに対して、その旨を表すセクションヘッダを付けます。これについては「**[Hack #33]　スタックに置かれたコードの実行を許可 / 禁止する**」で説明します。

まとめ

本 Hack では、GCC が生成したコードが実行中にコードを生成する場合について説明し、バッファオーバーフローといったセキュリティ上の問題との関連について解説を行いました。

—— Kazuyuki Shudo

HACK #33 スタックに置かれたコードの実行を許可/禁止する

ELF形式のオブジェクトにスタック上コードの実行可否を決めるフラグを埋め込むことで、スタックに置かれたコードの実行を許可することができます。

最近では、セキュリティ上の理由から、スタックやヒープに置かれたコードの実行はOSが禁止していることが一般的です。しかしそれを許可したい場合もあるため、GCC は ELF 形式のオブジェクトにスタック上コードの実行可否を決めるフラグを埋め込むことがあります。

スタック実行フラグの変更

次のコードでは、スタックに置かれたコード(トランポリン)を実行する必要が生じます。

```
void other(void (*funcp)()) {
  funcp();
}

void outer(void) {
  int a = 10;

  void inner(void) {
    printf("outer's a is %d\n", a);
  }

  other(inner);
}
```

これをtrampoline.cとして保存し、コンパイルします。適当なmain()関数を書いてouter()を呼び出すと、outer's... と表示され、何のことはなく実行できます。

続いて、得られたオブジェクトtrampoline.oのセクションヘッダをのぞいてみましょう。

```
% gcc -c trampoline.c
% readelf -S trampoline.o
...
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 7] .note.GNU-stack   PROGBITS        00000000 000105 000000 00   X  0   0  1
```

.note.GNU-stackというセクションヘッダのフラグ(Flg)がXとなっている点に注目してください。ここでXはexecute、つまり実行の可否を表します。

続いて、このフラグをリセットしてみます。ここではGNU binutilsのobjcopyコマンドを使います。prelinkコマンドでも同じことが可能です。すると、スタック上のコードを実行できなくなり、上記プログラムは異常終了するようになります。

```
% objcopy --set-section-flags .note.GNU-stack= trampoline.o trampoline-modified.o
(リンク)
(実行)
Segmentation fault
```

もしここで異常終了しないようなら、あなたがお使いのプロセッサやOSはスタックに置かれたコードの実行を禁止する機能を持たない、ということになります。

x86プロセッサとLinuxのexec-shieldパッチ

SPARCなどのプロセッサは以前より特定のメモリ領域を実行禁止にする機能を持っていました。しかし当初のx86プロセッサは、読み込みの可否と実行の可否が同一のフラグで表されており、読み込みを可能にすると自動的に実行も可能となってしまっていました。2004年初めごろにリリースされた更新版Pentium 4(Prescott)になってようやく、実行の可否を表す独立したフラグが導入されたという経緯があります。

ところが、Prescott以前のx86プロセッサでも、スタックやヒープ上のコードを実行禁止にできないわけではないのです。Linuxカーネル向けにexec-shieldというパッチが公開されていて、それを適用したカーネルでは、プロセッサがその機能を持たずとも、メモリ領域に対して読み込みは可能だが実行は不可という設定が可能となります。

Linuxディストリビューションによっては、カーネルにexec-shieldパッチを適用済み、という場合もあります(例:Fedora Core)。その場合、プロセッサに機能がなくとも、実行可否フラグを変更した上記のコードは異常終了するはずです。

まとめ

ELFオブジェクト中のスタック実行フラグを操作する方法を紹介しました。スタック上に置かれたコードの実行は、通常は、できないように設定されています。リンカ・ローダが.note.GNU-stackというセクションヘッダに対応している環境であれば、これを操作することで、スタック上コードの実行可否を制御できます。

—— Kazuyuki Shudo

HACK #34 ヒープ上に置いたコードを実行する

本Hackでは、mprotect(2)を使って、ヒープに置いたコードの実行を可能にする方法を紹介します。

mprotectシステムコール

メモリ保護がしっかりしているプロセッサやOSの上では、malloc(3)などで取得したメモ

リ領域、つまりヒープに置いたコードは実行できません。

次のプログラムを見てください。malloc(3)で確保したヒープに関数func()をコピーして、ヒープ上のコピーを呼び出すというプログラムです。これを実行すると、Segmentation Faultと表示されて異常終了してしまいます。異常終了しないとしたら、その環境ではこの種のメモリ保護が行われていないということです。

```
double func(void) {
  return 3.14;
}

int main(int argc, char **argv) {
  void *p = malloc(1000);
  memcpy(p, func, 1000);
  printf("PI equals to %g\n", ((double (*)(void))p)());
}
```

そこで、mprotect(2)の出番です。このシステムコールを使うことで、メモリ領域に対して許されるアクセス方法を変更することができます。上記プログラムに次の関数を追加し、

```
void allow_execution(const void *addr) {
  long pagesize = (int)sysconf(_SC_PAGESIZE);
  char *p = (char *)((long)addr & ~(pagesize - 1L));
  mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);
}
```

main()関数中のmemcpy()呼び出しのすぐ後に次の文を追加して、このallow_execution()を呼び出すようにします。

```
allow_execution(p);
```

これによって、異常終了せずに、PI equals ... と表示されるようになります。

mprotect(2)の引数

allow_execution()中では、sysconf(3)を使ってページ(OSがメモリを管理する単位)のサイズを取得しています。これは、mprotect(2)の第1引数をページサイズの倍数とするためです。少なくともLinux 2.4、2.6では、この引数はページサイズの倍数である必要があるようです。ある値の倍数から始まるメモリ領域を獲得するためには、posix_memalign(3)を使うという方法もあります。

ここではmalloc(3)で確保したヒープに対してmprotect(2)を呼び出しましたが、POSIXの規定では、mprotect(2)はmmap(2)で取得したメモリ領域に対してだけ使用できる、となっている点に注意してください。また、Mac OS Xではsysconf(3)ではなくマクロPAGE_SIZEで

ページのサイズを取得する必要があるかもしれません。

まとめ

本 Hack では、mprotect(2)を使ってヒープに置いたコードの実行を許可する方法を紹介しました。

—— Kazuyuki Shudo

HACK #35 PIE(位置独立実行形式)を作成する

PIE(位置独立実行形式)を作成する方法と特徴を紹介します。

通常、PIC(位置独立コード)は共有ライブラリに用いられますが、Linux上で最近のGCC、glibc、およびBinutilsを使うと、実行ファイルも位置独立にすることができます。本HackではPIE(位置独立実行形式)を作成する方法と特徴を紹介します。

PIE の基本

それでは例を見てみましょう。次のようなファイル foo.c があるとします。

```
#include <stdio.h>
void foo() {
    printf("hello\n");
}

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

このファイルを -fPIE というオプションを付けてコンパイルし、-pie というオプションをつけてリンクすれば PIE を作成できます。できあがったファイルは普通に実行できます。

```
% gcc -c -fPIE foo.c
% gcc -o foo -pie foo.o
% ./foo
hello
```

objdump -d で foo を逆アセンブルしてみると、アドレスが非常に小さい数字となっていることがわかります。これは、アドレス空間上のどの位置にマップしても動くよう、共有ライブラリと同様に、PIE 内はすべて相対アドレスになっているためです。

```
% objdump -d foo |grep main -A3
00000862 <main>:
 862:   55                      push   %ebp
 863:   89 e5                   mov    %esp,%ebp
 865:   83 ec 08                sub    $0x8,%esp
```

一方、gccにオプションを何も渡さずに作成した実行ファイルの場合、次のように絶対アドレスが割り振られています。

```
% gcc foo.c
% objdump -d a.out |grep main -A3
08048378 <main>:
 8048378:       55                      push   %ebp
 8048379:       89 e5                   mov    %esp,%ebp
 804837b:       83 ec 08                sub    $0x8,%esp
```

動的リンカld.soはPIEの実行時に、共有ライブラリに対して行うのと同様に、相対アドレスをアドレス空間上にマップする処理を行います。このとき、一部のLinuxディストリビューションではセキュリティ強化のためにマップするアドレスのランダム化を行っています。これには特定のアドレスを突くような攻撃を防ぐ効果があります。

実行も動的リンクもできるバイナリ

ここまで見てきたように、PIEは共有ライブラリと非常に似た性質を持っています。実際、リンク時にgccに-rdynamicオプションを加えると実行もできるし動的リンクもできるというバイナリを作成できます。-rdynamicは実行ファイルの中にも動的リンク用のシンボルを残すオプションです。gccに-rdynamicを渡すと、リンカldには--export-dynamicというオプションが渡されます。

次の例はファイルfooを動的リンクして関数foo()を呼び出すプログラムcall-foo.cを作成、実行しています。

```
% cat foo.c
#include <stdio.h>
void foo() {
    printf("hello\n");
}

% cat call-foo.c
void foo();
int main() {
    foo();
    return 0;
}
```

```
% gcc -c -fPIE foo.c; gcc -rdynamic -o foo -pie foo.o
% gcc -o call-foo call-foo.c ./foo
% ./call-foo
hello
```

実行も動的リンクもできるというバイナリがうれしい場面はあまりないと思いますが、非常に特殊な場面では役に立つかもしれません。下の例は自分自身を動的リンクしつつ階乗を計算するプログラムです。

```
% cat factorial.c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <assert.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        void *handle = dlopen ("./factorial", RTLD_LAZY);
        assert(handle != NULL);
        int (*factorial)(int) = dlsym(handle, "factorial");
        n = n *factorial(n - 1);
        dlclose(handle);
        return n;
    }
}

int main() {
    const int n = 5;
    printf("factorial(%d) = %d\n", n, factorial(n));
    return 0;
}

% ./factorial
factorial(5) = 120
```

本物の共有ライブラリとの違い

共有ライブラリと動的リンク可能なPICは似ていますが、非staticな関数の呼び出しに違いがあります。前者は共有ライブラリ内の非staticな関数をPLT経由で呼び出しますが、PIEはPIE内の非staticな関数はPLTを経由しないで直接呼び出します。

このため、PIEを動的リンクとして用いると、LD_PRELOADによるシンボルの置き換えの動作が共有ライブラリのときと変わってきます。LD_PRELOADはPLTを経由しない関数呼び出しに対しては効力がないためです。

まとめ

本Hackでは、PIE（位置独立実行形式）を作成する方法と特徴を紹介しました。現在、PIEはそれほど広く用いられている技術ではありませんが、セキュリティの強化にも使えるため、今後、利用される場面が増えていくのではないかと思います。

—— Satoru Takabayashi

HACK #36 C++ で synchronized method を書く

C++ で synchronized method を書く２つの方法を紹介します。

皆さんは、C++でマルチスレッドプログラムを書くとき、「同時に1つのスレッドにしか実行させたくない関数」をどのように記述するでしょうか？　本 Hack では、そのような関数の素敵な実装方法について解説します。

C の場合

C言語でそのような関数を実装する場合、おそらく次のようになるでしょう。これは特に問題ありません。正しく動作します。

```
void need_to_sync() {
  static pthread_mutex_t m = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
  pthread_mutex_lock(&m);
    // ... 何らかの処理 ...
  pthread_mutex_unlock(&m);
}
```

C++ の場合

では、C++ではこのような関数はどのようになるでしょうか？　C++ではたいていの場合、pthread_mutex_t型を直接使うのではなく、それをwrapしたMutexクラスを用い、pthread_mutex_lock 関数ではなく、Lock クラスを使うでしょう。

Lockクラスは、コンストラクタでMutexを受け取るとそれをすぐにロックし、デストラクタでロックを自動的に開放するようなクラスとして実装するのが通例です。RAII（Resource Acquisition Is Initialization）イディオムと呼ばれるこの方法を用いると、Mutexをロック中に例外がthrowされた場合でも、確実にMutexをアンロックすることができ、便利なわけです。

これから、C++ での 3 種類の実装例を見ていきます。

よくない方法（1）

すぐに思いつくのは、C 言語での例にかなり近い、次のような実装です。

```
#include <boost/thread/recursive_mutex.hpp>
void need_to_sync() {
  static boost::recursive_mutex m;
  boost::recursive_mutex::scoped_lock lk(m);
  // ... 何らかの処理 ...
}
```

しかし、これはうまく動きません。オブジェクトmのコンストラクタが呼ばれるのはneed_to_sync関数の初回呼び出し時なのですが(C++の規格でそう決まっています)、そのコンストラクタ呼び出しがスレッドセーフに行われる保証がないからです。

上記のコードをg++ -Sにかけてみましょう。すると、次のようなリストが得られます。わかりやすいようにコメントを付けました。

```
        # フラグが 0 でなければ .L11 へジャンプ
        cmpb    $0, guard variable for need_to_sync()::m
        jne     .L11
        # m のコンストラクタを呼び出す
        pushl   need_to_sync()::m
        call    boost::recursive_mutex::recursive_mutex()
        # フラグを 1 にする
        movb    $1, guard variable for need_to_sync()::m
.L11:
```

"guard variable for need_to_sync()::m"というのは、mのコンストラクタを呼び出したかどうかを覚えておくフラグです。上記をよく見ればわかるように、need_to_sync関数の初回呼び出しが複数のスレッドからほぼ同時に行われると、mのコンストラクタが複数回呼ばれるなどの誤動作を起こしてしまいます。この実装はうまく動きません。

よくない方法(2)

では、mを局所的な静的変数から非局所的な静的変数(グローバル変数やクラス変数)に変更したらどうでしょう？　GCCの場合、mのコンストラクタは、エントリポイントである_start関数経由で、main関数が呼び出させる前に呼び出されます(**[Hack #31]**)ので、いったん制御がmain関数に到達したあとは、need_to_sync関数は常に正常に排他制御を行うことができます。

```
#include <boost/thread/recursive_mutex.hpp>
namespace /* anonymous */ { boost::recursive_mutex m; }
void need_to_sync() {
  boost::recursive_mutex::scoped_lock lk(m);
  // ... 何らかの処理 ...
}
```

この方法は、たいてい問題なく動作するのですが、わかりにくい落とし穴が1つあります。

どこか他所の.cppファイルで非局所的な静的オブジェクトが使用されており、そのコンストラクタがneed_to_sync関数を直接または間接的に呼んでいると、need_to_sync関数は、まだコンストラクタによる初期化の済んでいないオブジェクトmをロックしてしまうことがあるのです。

mのような変数が複数ある場合に、その初期化順序を明示的に制御することは困難というわけです。この問題は、"static initialization order fiasco" と呼ばれています。もちろん Mutex だけでなく、下のようなコードも同様の問題を抱えていることになります。

```
const std::string FOO_BAR = "foobar"; // グローバル変数
```

詳しくは参考文献[1]を参照してください。

素敵な方法：C 互換構造体の Mutex を静的に初期化して用いる

"static initialization order fiasco"を回避する方法はいくつかありますが、ここでは最もバイナリアン的と思われる手法を解説しましょう。"static initialization order fiasco"が発生する原因は、要するにグローバルなオブジェクトを(コンストラクタなどで)動的に初期化していることが原因なわけですから、この動的な初期化をやめて、バイナリが実行された(mmap された)瞬間に初期化が終わっているようにすれば良いわけです。

C++の規格をよく読むと、「集成体」の一種である「C互換構造体」は静的に初期化できるとありますので、これを利用することにしましょう。C互換構造体は、コンストラクタ、デストラクタ、基底クラス、仮想関数、protected メンバ、private メンバなどを持てないという制約がありますが、非仮想関数を持つのは問題ありません。これがポイントです。Mutex/Lockの実装は次のようになるでしょう。static_mutexクラスは基底クラスや仮想関数を持てないため、scoped_lock クラスはテンプレートを使って static_mutex クラスの lock/unlock 関数を呼ぶようにしています。

```
// RAIIなロッククラスはテンプレートで実装
template<typename T>
class scoped_lock_ : private boost::noncopyable {
  T& m_;
public:
  explicit scoped_lock_(T& m) : m_(m) { m_.lock(); }
  ~scoped_lock_() throw() { m_.unlock(); }
};

// 静的に初期化可能なMutexクラス （エラー処理は省略）
#define STATIC_MUTEX_INIT {PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP}
struct static_mutex {
  pthread_mutex_t m_;
  void lock() { pthread_mutex_lock(&m_); }
  void unlock() { pthread_mutex_unlock(&m_); }
```

```
  typedef scoped_lock<static_mutex> scoped_lock;
};
```

これらを次のように使います。

```
namespace { static_mutex m = STATIC_MUTEX_INIT; }
void need_to_sync() {
  static_mutex::scoped_lock lk(m);
  // ...何らかの処理...
}
```

need_to_sync 関数を g++ -S にかけ、どんなコードが出力されるか見てみると……。

```
(anonymous namespace)::m:
        .long   0
        .long   0
        .long   0
        .long   1
        .long   0
        .long   0
```

このように、m の初期化内容(.long のデータ 6 つ)がオブジェクトに直接埋め込まれています。もちろん、動的な初期化は一切行われていません。ようやく、C++で問題なく synchronized method を実現することができました。

裏技：-fthreadsafe-statics

最近の GCC を使っている場合、実は「**よくない方法(1)**」のようなコーディングを行っても問題ない場合があります。g++ に -fthreadsafe-statics というオプションを与えて、「**よくない方法(1)**」のコードをコンパイルし、逆アセンブルしてみてください。

```
% g++ -D_REENTRANT -fthreadsafe-statics -c bad1.cpp
% objdump -Cdr bad1.o | grep -B1 __cxa_guard_acquire
118:   e8 fc ff ff ff          call   119 <need_to_sync()+0x1d>
                      119: R_386_PC32 __cxa_guard_acquire
```

もし、逆アセンブルしたリストの中で、上記のように__cxa_guard_acquireなる関数が呼ばれていたら、オブジェクトmのコンストラクタはGCCによってスレッドセーフな方法で呼ばれています。その GCC を使っているかぎりは「**よくない方法(1)**」のようなコードで synchronized method を実現しても問題ありません。

なお、-fthreadsafe-statics をサポートしている GCC の多くは、デフォルトで(暗に)このオプションが有効になっているようです。逆に、組み込み用途などでstatic変数のスレッドセーフな初期化による時間的、空間的なコストが気になる場合には、明示的に g++ -fno-

threadsafe-statics としたほうがよいでしょう。

まとめ

C++ で synchronized method を書くのは、予想以上に大変です。まとめると下の2つの方法があります。

- C互換構造体型のMutexクラスを自分で作成し、静的に初期化し、クラステンプレートを用いてロックする
- 何も工夫せずに既存の Mutex クラスを使う。ただし、g++ を -fthreadsafe-static オプション付きで使うことで様々な問題を回避する。

前者は複雑ですがクロスプラットフォームで、後者は特定バージョンのGCCに依存してしまいますが、お手軽です。

ところで、C++ 言語は、C 言語に比べるとずいぶん高水準な感じがしますが、C++ で今回のような際どいコーディングをする場合にも、binutilsはかなり役に立ちます。筆者は、規格に書かれている通りにC++のソースコードがコンパイルされたかどうか、objdump コマンド、nm コマンドを用いて逐一確かめながら作業しました。

参考文献

1. 「C++ FAQ Lite [10.12]～[10.16] What's the "static initialization order fiasco"?」（http://www.parashift.com/c++-faq-lite/ctors.html）
2. JIS X 3014:2003「プログラム言語 C++」（http://www.webstore.jsa.or.jp/）
   C++ 言語規格(ISO/IEC 14882:2003)の日本語訳です。

—— Yusuke Sato

HACK #37 C++ でシングルトンを生成する

マルチスレッド対応のシングルトンを４種類の方法で実装してみます。

本Hackでは、本書で紹介されている数々のHackを、C++というやや高級な言語でのコーディングで生かす方法を紹介します。具体的には、マルチスレッド対応のシングルトンを、4種類の方法で実装してみます。

シングルトンとは

シングルトンとは、「あるクラスに対してインスタンスが 1 つしか存在しないことを保証し、それにアクセスするためのグローバルな方法を提供する」というデザインパターンの 1 つです[1]。シングルトンをC++で実現する場合、コードはおおよそ次のような形になります。

```
class Singleton : private boost::noncopyable {
public:
  static Singleton *instance();  // 唯一のインスタンスを得る
private:
  Singleton(); // instance 関数以外からのオブジェクトの生成を禁止する
  static Singleton *instance_; // 唯一のインスタンスへのポインタ(最初は NULL)
};
```

シングルトンはシンプルでわかりやすいパターンですが、instance 関数が複数のスレッドから同時に呼ばれる可能性がある場合、実装上の細かな配慮が必要になります。instance 関数でスレッドセーフにシングルトンを生成するのは意外と難しいのです。

4 種類の実装

実際に、いくつかの方法でシングルトンを実装してみましょう。ここでは「**[Hack #36] C++ で synchronized method を書く**」で紹介した Hack、「静的に初期化できる Mutex」を使いますので、必要に応じて内容を理解しておいてください。

(1)もっとも保守的なシングルトン

まず、次のようなクラス変数 m_ を用意し、静的に初期化しておきます。

```
private:
  static static_mutex m_;
```

この Mutex を用いて、instance 関数を次のように実装します。

```
Singleton *Singleton::instance() {
  static_mutex::scoped_lock lk(m_);
  if(instance_ == NULL) instance_ = new Singleton;
  return instance_;
}
```

非常に分かりやすいのですが、一度オブジェクトが生成された後も、instance 関数が呼び出されるたびにMutexをロックしてしまいますので、あまり効率が良い方法とは言えません。

(2) メモリバリアとdouble checked lockingを用いたシングルトン

この点を改良するためによく使用されるのが、"Double Checked Locking"と呼ばれる、一定の条件で排他制御を省く方法です。

```
// よくない例
Singleton *Singleton::instance() {
  if(instance_ == NULL) {
    static_mutex::scoped_lock lk(m_);
    if(instance_ == NULL) instance_ = new Singleton;
  }
  return instance_;
}
```

ただし、冒頭部分(最初のif文)で、排他制御を行わずに複数スレッドで共有している変数(instance_)を操作していますので、「**[Hack #094] プロセッサのメモリオーダリングに注意**」で紹介している問題に十分に注意する必要がでてきます。このパターンをCPUやコンパイラによらずにうまく動作させるためには、次のようにメモリバリアを2つ、挿入する必要があるでしょう。詳細は「Double-Checked Locking, Threads, Compiler Optimizations, and More」[2]を参照してください。

```
// よい例 (参考文献 2 より引用・一部改変)
Singleton *Singleton::instance() {
  Singleton *tmp = instance_;
  RMB(); // メモリバリア
  if(tmp == NULL) {
    static_mutex::scoped_lock lk(m_);
    if(instance_ == NULL) {
      tmp = new Singleton;
      WMB(); // メモリバリア
      instance_ = tmp;
    }
  }
  return instance_;
}
```

RMB()やWMB()を実装する際には、例えばLinuxカーネルのlinux-2.6.x/include/asm-<アーキテクチャ>/system.hで#defineされているsmp_rmb()やsmp_wmb()を参考にすることができます。

```
// PowerPCでの例
#define RMB() __asm__ __volatile__ ("lwsync" : : : "memory")
#define WMB() __asm__ __volatile__ ("eieio"  : : : "memory")
```

なお、ここでメモリバリアの代わりにvolatile修飾を用いるのは、一般的には良いアイディアではありません。

(3) TLSを用いたシングルトン

「**[Hack #26]　TLS（スレッドローカルストレージ）を使う**」で紹介した__threadキーワードを利用して、高速なシングルトンを実装することもできます。

```
Singleton *Singleton::instance() {
   static __thread Singleton *tls_instance = 0;
   if (!tls_instance) tls_instance = do_instance();
   return tls_instance;
}

// private:
Singleton *Singleton::do_instance() { /* (1)と同じ */ }
```

各スレッドで一度だけ、(1)の方式（完全同期化）でインスタンスを取得するものの、そのインスタンスをTLSに保存してしまうことで、2回目以降のロックを不要にするわけです。お使いの環境でTLSが使用可能であれば、悪くない方法です。

(4) GCCの-fthreadsafe-staticsを用いたシングルトン

最近のGCCの機能を利用すると、非常に簡単にスレッドセーフなシングルトンを生成することができます。

```
Singleton *Singleton::instance() {
   static Singleton instance_;
   return &instance_;
}
```

クラス変数だったinstance_を、instance関数内部に移動すれば、あとは「**[Hack #36]　C++でsynchronized methodを書く**」で紹介したHackと同じ話です。最近のGCCを使っている場合に限定されるものの、instance_は高速かつスレッドセーフに初期化されます。

ベンチマーク

参考までに、4種類のシングルトンの速度を測定してみました。グラフは、筆者の環境（Linux/x86_64）でinstance関数を10億回呼ぶのにかかった時間です（図3-2）。(2)～(4)のどの方法が高速であるかは、環境によって異なります。

さらなるハック

本書のHackを活用すると、ここに挙げた(2)～(4)以外にも、様々なタイプのシングルトンを実装することが可能です。たとえば、「**[Hack #080]　自己書き換えでプログラムの動作を変える**」のHackを用いれば、「一度インスタンスが生成されたなら、instance関数を次のよ

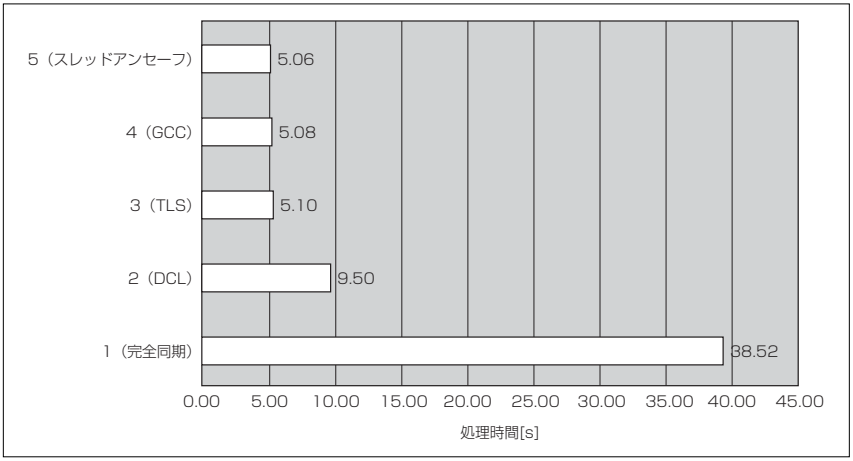


図3-2 4種類のシングルトンのベンチマークテスト

うな条件分岐なしのシンプルなものに自己書き換えしてしまう」という強烈なシングルトンも実現できるでしょう。

```
// 自己書き換え後
Singleton *Singleton::instance() {
  return instance_;
}
```

ぜひ色々と試してみてください。

まとめ

本Hackでは、C++の定番イディオムであるシングルトンを、4つの方法でスレッドセーフに実装しました。デザインパターンの実装にBinary Hackが役立つというのは、なかなか面白いと思います。

なお、ここでは「オブジェクトが本当に必要になった段階でそのオブジェクトを生成、初期化する」というタイプのシングルトンだけを扱いました。シングルトンの生成、破棄をどのように行うかを決めるのは難しい問題で、最適解は場合によって異なります。

参考文献

1.『オブジェクト指向における再利用のためのデザインパターン』
（エリック・ガンマ、ラルフ・ジョンソン、リチャード・ヘルム、ジョン・ブリシディース著、本位田真一、吉田和樹訳、ソフトバンククリエイティブ）

2 「Double-Checked Locking, Threads, Compiler Optimizations, and More」（http://www.nwcpp.org/Downloads/2004/DCLP_notes.pdf）
C++ での double checked locking の問題点について検討している文献です

—— Yusuke Sato

HACK #38 g++ の例外処理を理解する（throw 編）

g++ の throw が実際にはどのような処理になるかについて説明します。

C++ の例外処理は ANSI C だけでは書けない処理を含んでいます。

- ローカル変数のデストラクタを呼び出しつつ大域脱出
- 投げたオブジェクトの管理

本 Hack では、g++ がこれらを実際にはどう処理しているかについて解説します。

throw 処理

throw は実際には次のような処理を行います。

- __cxa_allocate_exception を呼び出して、例外オブジェクト用メモリを割り当てる
- __cxa_throw を呼び出して、例外ハンドラの検索と、ハンドラの呼び出しを行う

__cxa_throw、__cxa_allocate_exception はともに libstdc++ に含まれるライブラリ関数です。実装は GCC のソース中の libstdc++-v3/libsupc++/eh_throw.cc、eh_alloc.cc にあります。

例外処理の詳細については「**[Hack #39] g++ の例外処理を理解する（SjLj 編）**」で説明します。

例えば、以下のような関数は、

```
int func ( void )
{
        throw 0xff;
}
```

throw を使わないように書き直すと、以下のようになります。

```
#include <typeinfo>
#include <stddef.h>

extern "C" void __cxa_throw (void *thrown_exception, const std::type_info *tinfo,
```

```
void (*dest) (void *) );
extern "C" void *__cxa_allocate_exception( size_t size );

int func( void )
{
        void *throw_obj = __cxa_allocate_exception( sizeof(int) );      // - (1)
        *(int*)throw_obj = 0xff;                                        // - (2)
        __cxa_throw( throw_obj, &typeid(int), NULL );                   // - (3)
}
```

(1)の部分で、投げるオブジェクトの領域を割り当てています。__cxa_allocate_exceptionの引数は、割り当てるサイズです。

(2)の部分で、投げるオブジェクトの値を設定しています。ここでは、投げるオブジェクトはint型の値0xffなので、そのように設定しておきます。

(3)の部分で、実際の投げる処理を行っています。__cxa_throwの引数は、それぞれ、投げるオブジェクト、そのオブジェクトの型情報、投げたオブジェクトの後始末を行う関数へのポインタです。なお、g++では、後始末はcatchする部分で行われることから、ここでは最後の引数はNULLにしておきます。

Itanium C++ ABI

これらの例外処理のインターフェースは、Itanium C++ ABI（http://www.codesourcery.com/cxx-abi/abi.html）と呼ばれるABIの例外処理の部分（http://www.codesourcery.com/cxx-abi/abi-eh.html）で決められています。

まとめ

throwが実際にはどのような処理になるかについて説明しました。

throwは実際には、GCCに含まれるlibstdc++に含まれるライブラリ関数を呼び出して実現しています。

—— Takashi Nakamura

HACK #39

g++の例外処理を理解する（SjLj編）

g++のSjLjを使った場合の例外の実装について説明します。

C++例外処理をするにあたって、大域脱出の仕組みは不可欠です。g++では、この大域脱出を実現するために、それぞれ、Unwind-SjLj、Unwind-dw2と呼ばれる2種類の大域脱出の仕組みを用意しています。

ここでは、Unwind-SjLjについて説明します。Unwind-SjLjは、Cygwin、Mingw環境で使わ

れています。また、GCCのビルド時にconfigureに--enable-sjlj-exceptionsオプションを渡すと、他の環境でも使うことができます。

手元のGCCがどちらのSjLjを使っているのかは、以下のようにして調べることができます。

```
$ gcc -v
```

出力の中に--enable-sjlj-exceptionsの文字列があった場合、そのコンパイラはSjLjを使っています。

なお、以下のプログラムの動作確認は--enable-sjlj-exceptions付きでコンパイルしたGCCのバージョン 3.4.4 を i686 の Linux 上で動かして行っています。

Unwind-SjLj を理解する

Unwind-SjLj の実装はおおよそ下のようになっています（実装は GCC のソースの gcc/unwind.inc にあります。_Unwind_RaiseException にてハンドラの探索が行われ、_Unwind_RaiseException_Phase2 にてクリーンアップが行われるようになっています）。

- catch するかどうかを判定する関数（personality と呼ばれます）と、catch した時に、どこに longjmp するかを記録した jmp_buf を一緒にした SjLj_Function_Context をリストにつないでおく。
- throwすると、SjLj_Function_Contextのリストをたどっていく。そのとき、personality を使って、捕まえられる型かどうかの判定を行う。
- 捕まえられる型だった場合、再度、SjLj_Function_Contextのリストをたどっていって、後始末を行う。
- 後始末が終わったら、SjLj_Function_Context に含まれる jmp_buf へ longjmp する。

まず、実際のcatch処理とは違いますが、SjLjを使った大域脱出を理解するために、とりあえず、catch するだけのプログラムを紹介します。

```
#include <unwind.h>
/* unwind.h は、大域脱出に関するいくつかの型定義とプロトタイプ宣言を含むヘッダです */

#include <setjmp.h>
#include <stdio.h>

struct SjLj_Function_Context  /* SjLj_Function_Contextの定義は gcc/unwind-sjlj.c にあります */

{
```

```
    struct SjLj_Function_Context *prev;
    int call_site;
    _Unwind_Word data[4];
    _Unwind_Personality_Fn personality;
    void *lsda;
    jmp_buf jbuf __attribute__((aligned));
};

static _Unwind_Reason_Code my_personality(int code,
                      _Unwind_Action act,
                      _Unwind_Exception_Class cls,
                      struct _Unwind_Exception *e,
                      struct _Unwind_Context *ctxt ) {
    // catch できるかどうかの判定。ここでは、常に catch できるものとしておきます。

    if ( act & _UA_CLEANUP_PHASE )
        return _URC_INSTALL_CONTEXT;
    else if ( act & _UA_SEARCH_PHASE )
        return _URC_HANDLER_FOUND;
    /* とりあえずここまでこないはず */
    return (_Unwind_Reason_Code)0;
}

void catch_func( void ) {
    struct SjLj_Function_Context sjlj;
    sjlj.personality = my_personality;
    sjlj.lsda = NULL;
    if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
        puts("catch");
    } else {
        _Unwind_SjLj_Register( &sjlj );  // ここで sjlj をリストにつないでいる
        throw 0;
    }
    _Unwind_SjLj_Unregister( &sjlj );
}
int main( ){ catch_func(); }
```

_Unwind_SjLj_Register を呼び出すと、sjlj がリストにつながれます。

throwすると、内部でpersonalityが引数_UA_SEARCH_PHASE付きで呼ばれ、catchできるかどうかの判定が行われます。ここでは、引数が_UA_SEARCH_PHASEだった場合、常に、catchできるものとして、_URC_HANDLER_FOUNDを返すようにしておきます(personalityについては次に詳しく説明します)。

catch できると判定されたので longjmp して、setjmp したところへ帰ってきます。

最後に、後始末として、_Unwind_SjLj_Unregisterを呼び出して、SjLj_Function_Contextのリストから sjlj を、取り除いておきます。

このプログラムでとりあえず、catch だけはできました。

personality

g++が使う大域脱出の仕組みは、実は、C++専用のものではありません。様々な言語で対応できるよう、以下の処理が分離されています。

- 大域脱出する場合の後始末
- 対応するハンドラかどうかのチェック

SjLj_Function_Contextのメンバであるpersonalityを変更することによって、この処理を置き換えることができます。

対応するハンドラかどうかチェックする場合は、2番目の引数に_UA_SEARCH_PHASEを渡して、personalityが呼ばれます。このとき、personalityが_URC_HANDLER_FOUNDを返せば、対応するハンドラとされます。

後始末する場合は、2番目の引数に_UA_CLEANUP_PHASEを渡して、personalityが呼ばれます。このとき、personalityが_URC_INSTALL_CONTEXTを返せば、そこで後始末を終了し、大域脱出が実際に行われることになります。

上のmy_personalityでは、特に何もしていませんが、実際のC++の例外処理として使う場合には、このpersonalityが呼ばれたときに、以下の処理が行われなくてはいけません。

- catchできる型かどうかのチェック
- ローカル変数のデストラクタ呼び出し

g++では、C++の例外処理に対応したpersonalityとして、__gxx_personality_sj0というライブラリ関数を用意しています（__gxx_personality_sj0の実装は、GCCソースのlibstdc++-v3/libsupc++/eh_personality.ccにあります）。

Language Specific Data Area

__gxx_personality_sj0でcatchできる型かどうかのチェックを行う、というところまではよいのですが、上のプログラムのcatch_func関数では、catchできる型の情報についてはまったく書かれていません。catchする型の情報はどのような形で用意しておけばよいのでしょうか？

__gxx_personality_sj0では、catchできる型かどうかの情報は、LSDA（Language Specific Data Area）と呼ばれるところに入れることになっています。LSDAの構造は以下のようになっています（SjLjを使ったg++が生成するLSDAの場合です。実際は、LSDAの構造は使い方によって変化します）。

```
struct lsda {
    unsigned char lpstart_format;  // landing_pad 開始アドレスのフォーマット
    unsigned char ttype_format;    // 型情報アドレスのフォーマット

    unsigned char type_offset;     // 型情報までのオフセット

    unsigned char call_site_format; // call-site データのフォーマット
    unsigned char call_site_length; // call-site の大きさ

    /* landing_pad、action_record_table、catch_type の大きさは可変です */
    unsigned char call_site_table[ call_site_length*2 ];
    signed char action_record_table[N];

    const std::type_info *catch_type[N];
} __attribute__((packed)) ;
```

アドレスのフォーマットはpcrel(0x10:PCからの相対位置)、absptr(0x00:絶対アドレス)などが指定できます。特に指定しない場合は0xff(omit)となります。

通常、g++が生成したLSDAの場合はlpstart_format=0xff、ttype_format=0x00となります。

type_offsetは、型情報の相対位置です。type_offsetの終わりから見た、catch_typeの終わりの位置までのオフセットが入ります。

call-siteのデータフォーマットは、符号なし4バイト、符号付き2バイトなど、選択することができます。g++が生成したLSDAではuleb128になっていました。uleb128は、DWARFで使われている可変長データの表現です。8bit(1byte)のうち7bitをデータとして使い、1bitを次にデータが続くかどうかのフラグとして使います。符号付きの場合が、sleb128、符号なしの場合がuleb128になります。

call_site_tableには例外が起きた位置と、どのaction_recordを使うのかの対応が入れられています。

```
try { t(); } catch ( int x ) { return x;}
try { t(); } catch ( char x ) { return x; }
```

このような場合は、int用のaction_tableとchar用のaction_tableと、それを対応付けるためのcall_site_tableが作られます。

action_record_tableは、ハンドラ側でどの型がキャッチされたかを区別するのに使われます。型情報と識別番号が対応するようにしておきます。

以上をもとに、LSDAを定義し、最初のプログラムを__gxx_personality_sj0を使うように書き換えてみます。まず、LSDAを定義します。

```
struct lsda {
    unsigned char start_format; // 0
    unsigned char type_format; // 1
```

```
    unsigned char type_offset; // 2
    unsigned char call_site_format;// 3
    unsigned char call_site_length;// 4
    unsigned char call_site_table[2]; // 5
    signed char action_table[2]; // 7
    const std::type_info *catch_type[1]; // 9
} __attribute__((packed)) ;
static struct lsda my_lsda = {
    0xff,
    0x00,
    10,   // type_offsetの終わり(3)から、catch_typeの終わり(13)までのオフセット
    0x01,
    2,
    { 0,1 },
    { 1, 0 },
    &typeid( int ), /* 捕まえる型 */
};
extern "C" _Unwind_Reason_Code __gxx_personality_sj0( int, _Unwind_Action,
                                _Unwind_Exception_Class, struct _Unwind_Exception *,
                                struct _Unwind_Context * );
```

Unwind-SjLjを利用する場合は、これをSjLj_Function_Contextのlsdaメンバに入れておきます。

```
< sjlj.personality = my_personality; /* これを変更 */
< sjlj.lsda = NULL;

> sjlj.personality = __gxx_personality_sj0;
> sjlj.lsda = (void *)&my_lsda;
> sjlj.call_site = 1;
```

また、sjlj.call_siteで、call_site_tableのうち、何番目のcall_siteを使うのかを指定します。この場合、call_siteは1個しかないので、1を指定しておきます(call_siteのインデックスは1オリジンになります)。

これで、g++が使うpersonalityが呼ばれるようになり、型判定と途中のデストラクタ呼び出しが行われるようになりました。

catchしたオブジェクトの扱い

catchしたオブジェクトは以下のようにして取得できます。

```
if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
    void *thrown_obj = __cxa_begin_catch( (void*)sjlj.data[0] );
    printf("%d\n",*(int*)thrown_obj);
    __cxa_end_catch( );
}
```

まず、setjmpした場所に帰ってきたとき、sjljのメンバdataの0番目に、例外に関する内部情報を入れたオブジェクトが入れられています。__cxa_begin_catchを呼び出すと、この内部情報をもとに処理が行われたあと、投げられたオブジェクトを返してきます（__cxa_begin_catchの実装は、libstdc++-v3/libsupc++/eh_catch.ccにあります）。

ここで返ってくるオブジェクトは、「**[Hack #39] g++の例外処理を理解する（throw編）**」で説明した、__cxa_allocate_exceptionで割り当てたオブジェクトになります。

最後に、後始末をするために、__cxa_end_catchを呼び出しておきます。__cxa_end_catchは、例外を扱う場合に一時的に割り当てられたオブジェクトと投げられたオブジェクトの解放を行います。

catchを行うプログラム

これで、catchの判定から終了までを行うことができるようになりました。以上より、

```
void catch_func( void ) {
    try {
        throw 100;
    } catch ( int x ) {
        printf("%d\n",x);
    }
}
```

というプログラムをcatchを使わないで書きなおすと、以下のようになります。

```
#include <unwind.h>
#include <setjmp.h>
#include <stddef.h>
#include <stdio.h>
#include <typeinfo>

struct SjLj_Function_Context
{
    struct SjLj_Function_Context *prev;
    int call_site;
    _Unwind_Word data[4];
    _Unwind_Personality_Fn personality;
    void *lsda;
    jmp_buf jbuf __attribute__((aligned));
};

struct lsda {
    unsigned char start_format; // 0
    unsigned char type_format; // 1
    unsigned char type_length; // 2
    unsigned char call_site_format;// 3
    unsigned char call_site_length;// 4
```

```
    unsigned char call_site_table[2]; // 5
    signed char action_table[2]; // 7
    const std::type_info *catch_type[1]; // 9
} __attribute__((packed)) ;
static struct lsda my_lsda = {
    0xff,
    0x00,
    10,
    0x01,
    2,
    { 0,1 },
    { 1, 0 },
    &typeid( int ), /* 捕まえる型 */
};
extern "C" _Unwind_Reason_Code __gxx_personality_sj0( int, _Unwind_Action,
                                _Unwind_Exception_Class, struct _Unwind_Exception *,
                                struct _Unwind_Context * );
extern "C" void *__cxa_begin_catch (void *exc_obj_in) throw();
extern "C" void __cxa_end_catch ();

void catch_func( void ) {
    struct SjLj_Function_Context sjlj;

    sjlj.personality = __gxx_personality_sj0;
    sjlj.lsda = (void *)&my_lsda;
    sjlj.call_site = 1;

    if ( __builtin_setjmp(sjlj.jbuf) == 1 ) {
        void *thrown_obj = __cxa_begin_catch( (void*)sjlj.data[0] );
        printf("%d\n",*(int*)thrown_obj);
        __cxa_end_catch( );
    } else {
        _Unwind_SjLj_Register( &sjlj );
        throw 100;
    }

    _Unwind_SjLj_Unregister( &sjlj );
}

int main( ) {
    catch_func( );
}
```

まとめ

SjLj を使った場合の例外の実装について説明しました。

例外処理はオブジェクトが飛んでいくような不思議な現象ではなく、地道に処理される普通のプログラムであることがわかるはずです。

—— Takashi Nakamura

HACK #40 g++ の例外処理を理解する(DWARF2 編)

g++ のDWARF2 の情報を使った大域脱出と、それを使った例外処理について説明します。

Unwind-dw2は、デバッグ情報用のフォーマットであるDWARF2を使った大域脱出の仕組みです。Unwind-dw2を使うことによって、例外が発生しないかぎり実行時コストがほぼゼロの大域脱出を実現することができます。コストについては、「**[Hack #41] g++ 例外処理のコストを理解する**」も参照してください。

ここでは、DWARF2の情報を使った大域脱出と、それを使った例外処理について説明します。

DWARF2

DWARF2(Debug With Arbitrary Record Format Version 2)は、デバッグ用に使われる情報フォーマットの仕様です。仕様は、http://dwarf.freestandards.org/ で入手することができます。

DWARF2では、型、ファイル位置、フレーム情報などのフォーマットが決められています。Unwind-dw2は、これらのうちのフレーム情報を利用します。g++は、コードを生成するときに、DWARF2のフォーマットに従って生成したコードのフレーム情報も一緒に生成します。g++ でプログラムをコンパイルすると、.eh_frame_section 中に .LSFDExx や .LSCIExx などのラベルを見つけることができます。これが、生成されたフレーム情報になります。

このフレーム情報は、プログラムカウンタをもとに、レジスタの状態やスタックフレームの状態を取得できるようになっています。スタックフレームの状態がわかれば、関数からの戻りアドレスがわかります。プログラムカウンタから、フレームの状態が取得できるので、戻りアドレスがわかれば、さらにそこから、関数が呼び出される前のスタックの状態を取得できます。これを繰り返していけば、関数呼び出しを巻き戻していくことができます。

Unwind-dw2による大域脱出は、このフレーム情報による巻き戻しを行うことで実現されています。

DWARF2 のフレーム情報

そのDWARF2 のフレーム情報はどのようになっているかについて簡単に説明しておきます。

DWARF2のフレーム情報は、バイトコードプログラムになっています。このバイトコードは、コンパイル時にオプション -s、-dA を付けて、出力されたアセンブリを見ればコメント付きでバイトコードを見ることができます。

```
.byte    0x4    # DW_CFA_advance_loc4
.long    .LCFI0-.LFB2
.byte    0xe    # DW_CFA_def_cfa_offset
.uleb128 0x8
.byte    0x85   # DW_CFA_offset, column 0x5
.uleb128 0x2
.byte    0x4    # DW_CFA_advance_loc4
.long    .LCFI1-.LCFI0
.byte    0xd    # DW_CFA_def_cfa_register
.uleb128 0x5
```

このように、DW_CFA_で始まる名前が命令になっています。このプログラムが、レジスタの状態、フレームの状態を更新していくことで、フレームの構造が実行時にわかり、戻りアドレスなどで大域脱出の情報を得ることができます。インタプリタの実装は、gcc/unwind-dw2.cにあります。

フレーム情報のソート

実際に大域脱出処理を行うには、プログラムカウンタからフレーム情報を検索する必要があります。ここで、二分探索を行うために、フレーム情報はあらかじめソートされるようになっています。

フレーム情報のソートは、リンク時に行われます。ldに--eh-frame-hdrオプションを渡すと、ldは、eh_frameセクションを調べ、フレームの開始アドレス順にソートした結果を、.eh_frame_hdrセクションに埋め込みます。この実装は、binutilsのbfd/elf-eh-frame.cにあります。

例外テーブル

ここまでの解説により、DWARF2のフレーム情報があれば、スタックの巻き戻しを行い、大域脱出を行えることがわかっていただけたはずです。ここからは、Unwind-dw2を使った場合に例外処理はどのように実現されているかについて説明します。

Unwind-dw2を使った場合も、例外処理の大体の流れはUnwind-SjLjと同じです。personalityが呼び出されて、型の判定と、デストラクタの呼び出し、そして、最後に例外ハンドラが実行されます。しかし、Unwind-dw2は、ハンドラの検索方法が、Unwind-SjLjを使った場合と異なります。Unwind-dw2を使った例外処理では、例外テーブルを用いて、ハンドラの検索が行われるのです。

例外テーブルとは、例外が投げられた場所と、例外ハンドラのアドレスの対応を記録したテーブルです。例えば、次のプログラムの場合、

```
int func() {
    try {
        statements...;                      -+
        try {           -+                   |
            func();      | (1)               |
            ...;        -+                   | (2)
        } catch ( Obj *p ) { return func(); } |
                                              |
        statements...;                      -+
    } catch ( int i ) {
        return i;
    }
}
```

(1)の中で、Obj* 型の例外が投げられた場合、func()を実行、(2)の中で int 型の例外が投げられた場合、return i を実行となり、例外テーブルには、その対応が記録されます。

次のプログラムをコンパイルして、以下のようなhogehoge関数呼び出しを探してください。

```
extern "C" void hogehoge();
void func( ) {
    try { hogehoge(); } catch ( int i ){}
}

.LCFI2:
.LEHB0:
    call    hogehoge
.LEHE0:
```

ここで関数呼び出しは、.LEHB0 から .LEHE0 の区間で行われています。例外テーブルは、.gcc_except_table セクションにLSDAの一部として存在しています。.gcc_except_table を探すと、以下のようなものが見つかるはずです。

```
    .section    .gcc_except_table,"a",@progbits
    .align 4
.LLSDA2:
    .byte    0xff
    .byte    0x0
    .uleb128 .LLSDATT2-.LLSDATTD2
.LLSDATTD2:
    .byte    0x1
    .uleb128 .LLSDACSE2-.LLSDACSB2
.LLSDACSB2:                 ; LFB2 は 関数の最初です
    .uleb128 .LEHB0-.LFB2   ; 例外をキャッチする範囲の始めの位置までのオフセット
    .uleb128 .LEHE0-.LEHB0  ; 例外をキャッチする範囲の大きさ
    .uleb128 .L7-.LFB2      ; 例外ハンドラのオフセット
    .uleb128 0x1            ; int の型情報(_ZTIi)参照用
    .uleb128 .LEHB1-.LFB2   ; これは g++ が生成したライブラリ関数(_Unwind_Resume 呼び出し)用の
テーブル
    .uleb128 .LEHE1-.LEHB1
```

```
    .uleb128 0x0
    .uleb128 0x0
.LLSDACSE2:
    .byte    0x1
    .byte    0x0
    .align 4
    .long    _ZTIi
.LLSDATT2:
```

ここで、例外をキャッチする範囲と、そのハンドラのアドレスの対応が記録されているのがわかります。

まとめ

Unwind-dw2 の例外処理は、次のように実装されています。

- プログラムカウンタからフレーム情報を取得
- フレーム情報に含まれる LSDA から例外テーブルを取得
- 投げられた型に対応するハンドラがあるかチェック。ある場合はそのハンドラを実行
- 投げられた型に対応するハンドラがない場合は、フレーム情報をもとに、呼び出し元アドレスを取得
- 呼び出し元アドレスのフレーム情報を取得
- 以上をキャッチできるまで繰り返す

フレーム情報と例外テーブルは、コンパイル時に静的に決定することができます。Unwind-dw2を使った例外処理は、このようにして、「例外が発生しないかぎりコストはほぼ0」を実現しています。

—— Takashi Nakamura

HACK #41

g++ 例外処理のコストを理解する

さまざまな場合に応じて、例外処理に必要なコストを解説します。

例外処理には、多少のコストが必要になります。ここでは、さまざまな場合に応じて、どのぐらいのコストが必要になるかについて説明します。

関数を呼ばない、try-catch を使わない関数はコスト 0

```
void func( int x ) { return x + 3; }
```

この場合、サイズ、実行時間ともにコストはありません。

例外を投げる関数を呼び出すと関数ごとにフレーム情報が追加される

```
extern void f();
void g() { f(); }
```

この場合、DWARF2を使って例外処理を行っている場合、スタック巻き戻しに必要なフレーム情報が追加されるため、サイズが増加します。実行時間は変わりません。SjLjを使って例外処理を行っている場合、および以下のように関数が例外を投げないことが明示的に記述されている場合、サイズ、実行時間ともにコストは変わりません。

```
extern void f() throw ();
void g() { f(); }
```

try ブロック内で例外が発生する可能性がある場合、catch のための処理が追加される

```
extern void f();
void g() { try { f() } catch ( ... ) { ... } }
```

この場合、catchのためのコストが発生します。SjLj、DWARF2のどちらを使った場合にも、キャッチする型を指定する必要があるので、LSDA(Language Specific Data Area)が必要になります。その分のサイズが増えます。

SjLj を使っている場合

try-catchを通るたびにSjLj_Function_Contextのセットアップ処理が行われるようになります。SjLj_Function_Contextをスタック上に確保するためにスタック領域を消費します。

DWARF2 を使っている場合

SjLjと違い、セットアップ処理が不要なので、例外が発生しない場合(try節を通るだけの場合)は、実行時間は殆ど変わりません(ハンドラを越えるためのジャンプ命令が増えるだけです)。例外テーブルが必要になるので、その分SjLjと比べて、LSDAのサイズが増えます。

try-catch と同等のコストが必要になる場合

デストラクタ呼び出しが必要になる場合

```
struct C { ~C(); } ;
extern void t();
void func() { C c; t(); }
```

このプログラムは実際には、下のような処理になるため、try-catchと同じコストが必要になります。

```
void func () {
    try { t();}
    catch (...) { c.~C(); throw; }
    c.~C();
}
```

throw() 付きの関数で例外を投げるかもしれない関数を呼んだ場合

```
void t();   // 例外を投げるかもしれない

void func() throw () {
    t();
}
```

このプログラムは実際には、下のような処理になるため、try-catchと同じコストが必要になります。

```
void func() {
    try { t(); } catch (...) { std::__cxa_call_unexpected( ... ) }
}
```

例外が起こった場合の処理時間は、SjLj のほうが速い

Unwind-SjLjでは、tryする場所で設定したSjLj_Function_Contextをたどっていくだけです。

しかし、Unwind-dw2で大域脱出するには、スタックフレームをたどって、フレーム情報を1個ずつ調べながら、スタック巻き戻しを行う必要があります。tryしたかどうかに関わらず、通ってきた関数をすべて処理することになるのです。また、Unwind-dw2では、プログラムカウンタからフレーム情報を検索する必要があります。

```
#include <stdio.h>
int t() { throw 0xff; }

int recursive( int i ) {
    if ( i==0 ) { t(); }
    else recursive( i-1 );
```

```
}

#define N 65535
int c( int n ) {
    int i;
    for ( i=0; i<N; i++ ) {
        try { recursive(n); }
        catch ( ... ) {}
    }
}

int main() {
    c( 3 );
    return 0;
}
```

このようなプログラムで実験したところ、筆者の環境では、dw2を使ったほうが約1.3秒、SjLjのほうが、約0.04秒となりました。

また、再帰の深さを変化させた場合、SjLjでは時間はほとんど変化しませんが、dw2のほうは、再帰を深くした分だけ、遅くなります。c(3) -> c(30)の場合で、SjLjが約0.04秒、dw2のほうが約5.67秒となりました。

まとめ

DWARF2のほうは例外が発生しない限り、例外が存在しない場合とほぼ同じスピードで実行されます。SjLjのほうは、若干のオーバーヘッドがあります。例外が発生した場合の処理はSjLjのほうがDWARF2よりも高速です。

—— Takashi Nakamura

4章

セキュアプログラミング Hack

Hack #42-57

現在、セキュアなプログラムを書くことはプログラマにとって最も重要な課題の１つになっています。本章では、CおよびC++でセキュアなプログラムを書くためのさまざまな実践的なテクニックを紹介します。まずはじめにGCCが提供する各種のセキュリティ強化機能を紹介し、次にC/C++のプログラムを書く上で気を付けるべき注意事項を、最後にセキュアなプログラムを書く上で役立つツールを紹介します。

GCCの提供するセキュリティ機能はGCCのバージョンが上がるたびに着実に進歩しています。また、Valgrindはメモリ関連のバグを検出するツールの定番です。これらのテクニックを知っていると知らないのでは、セキュアなプログラムを書く上で大きな差となります。

HACK #42 GCC セキュアプログラミング入門

基本中の基本である、GCCの警告オプションや__attribute__の活用法を解説します。

ネットワークに接続したり、個人情報を扱ったりするソフトウェアが増えてきたことで、セキュリティを考慮したプログラミングが、多くのプログラマに必要とされるスキルになってきました。本章ではセキュアプログラミングに関連するHackを紹介します。

本章の内容

ソフトウェアのセキュリティ欠陥は、「認証機能の入れ忘れ」に代表されるように、要件定義レベルのミス、マクロな視点でのミスによって混入する場合もありますが、もちろんコーディング時にも混入します。「悪魔は細部（膨大なソースコードの中のたった数行）にも潜む」といったところでしょうか。

ところで、膨大なソースコードからバッファオーバーフロー、整数オーバーフロー、各種レースコンディションといった微妙な問題を見つける作業や、そもそもその種の問題を起こさないようにセキュアなコーディングを行う作業は大変なことが多いものですが、逆に、CPU、OS、バイナリフォーマット、言語処理系などを知り尽くしたバイナリアンが、その能

力を存分に発揮できる領域でもあります。

前提知識

本章のHackのいくつかは、GCCとglibcの使用を前提としたものです。ここでは、セキュリティを考慮したコーディングを行う際の、GCC と glibc の基本的な使い方を解説します。

GCC の警告オプション

セキュアなコードを書くための最初の一歩は、コンパイラの警告に耳を傾けることでしょう。セキュリティに配慮したコードを書く際には、次の警告オプションを有効にすることをおすすめします。

`-Wall`
: 一般的に有益と思われる警告をすべて表示します。

`-W`
: -Wall では表示されない、いくつかの警告を追加で表示します。このオプションは、GCC4 で -Wextra に名前が変わりました。

`-Wformat=2`
: printf関数などの書式指定文字列が、文字列リテラルではない場合に警告します。これは、"format-string bug"と呼ばれるセキュリティホールの発見に役立つ場合があります。書式指定文字列を変数で与えるのはなるべく避けましょう。

`-Wstrict-aliasing=2`
: C言語の規格上、許されていない方法でのメモリアクセスを警告します。-O2と共に使用します

strict-aliasing rule

-Wstrict-aliasing=2 は、オーバーラップする 2 つのメモリ領域を、異なる型 A、B で読み書きした場合などに警告を表示するオプションです。このような読み書きはC言語の規格で禁止されていますので、意図的に行う場合を除けば、なるべく避けたいものです。-Wallだけでもある程度は警告されますが、-Wstrict-aliasing=2 を明示的に指定すると、チェックがより厳しくなります。

```
int main(int argc, char **argv) {
  /* short(=16bit)で書きこんで */
  ((short*)&argc)[0] |= 1;
```

```
  ((short*)&argc)[1] |= 1;
  /* int(=32bit)で読む */
  printf("%d\n", argc);
  return 0;
}
```

このコードは、最適化によって動作が変化します。これは、最適化時にGCCが変数の読み書きを reorder（並べ替え）することが原因です。どのように並べ替えられるかは、gcc -S や objdumb -d で確認してください。

```
% gcc     alias.c && ./a.out
65537
% gcc -O2 alias.c && ./a.out
1
```

警告オプションを適切に使用することで、この問題を検出することができます。

```
% gcc -O2 -Wstrict-aliasing=2 alias.c
alias.c:4: warning: dereferencing type-punned pointer will break strict-aliasing
rules
```

GCC の警告オプション（使用例）

参考までに、筆者は普段、下のようなオプションを使用しています。

```
% gcc4 -Wall -Wextra -Wformat=2 -Wstrict-aliasing=2          \
       -Wcast-qual -Wcast-align -Wwrite-strings -Wconversion \
       -Wfloat-equal -Wpointer-arith -Wswitch-enum foo.c
```

g++ の場合は、これに加えて -Woverloaded-virtual や -Weffc++ を使用することもあります。また、キャストがたくさん使われている C/C++ コードをコンパイルする際は、**-O2** の後に -fno-strict-aliasing を指定し、最適化を若干弱めるようにもしています。

詳しくは、GCC のマニュアルを参照してください。

GCC の __attribute__((format))

printf 関数互換の書式指定文字列を受け取る関数を自作する場合、「**[Hack #22]** GCC の GNU **拡張入門**」で紹介されている "format" という attribute を活用しましょう。

```
__attribute__((format(printf, 1, 2))) void my_printf(const char *my_format, ...) {
  va_list ap;
  assert(my_format != NULL);
  va_start(ap, my_format);
    vprintf(my_format, ap);
    vsyslog(LOG_ERR, my_format, ap);
```

```
    va_end(ap);
}
```

ここで、数字の1は関数の引数の中の書式指定文字列の位置、2は可変長引数の開始位置を表します。このようにしておくと、-Wformat=2による警告が、自作のprintf系関数にも適用されるようになり、format-string bugの発見率が向上します。

glibcによるHeap Consistency Checking

glibcは、間違ったrealloc/free関数の使用をランタイムにチェックする機能をデフォルトで備えています。例えば、mallocで確保したメモリを2度freeした場合に、それを検出することができます。

```
% gcc -Wall -Wextra -Wformat=2 -Wstrict-aliasing=2 -o double_free *.c
% MALLOC_CHECK_=1 ./double_free
malloc: using debugging hooks
*** glibc detected *** free(): invalid pointer: 0x08d04008 ***
```

コンパイルは通常通り行えばよく、実行時に環境変数MALLOC_CHECK_で、このチェック機能の有効無効を切り替え可能なのがポイントです。MALLOC_CHECK_に指定できる数値は次の通りです。

MALLOC_CHECK_の値	異常検出時の動作
1	stderrに警告文を表示し実行を継続する
2	即abortする

コマンドラインから実行する場合は1、gdb上でデバッグを行う際は2が良いでしょう。

```
% gdb ./double_free
(gdb) set environment MALLOC_CHECK_=2
(gdb) run
...
```

メモリの二重開放(double free bug)は、誤動作やセキュリティホールの原因になることがありますので、このチェックは有益です。

まとめ

セキュアなコードを書くのはなかなか大変な作業ですが、GCCやglibcの機能をフル活用することで、すこし楽ができます。本Hackでは、まず基本中の基本として、GCCの警告オプションや__attribute__の活用法を解説しました。

参考文献

- 「Secure Coding in C and C++」（http://www.cert.org/books/secure-coding/）
  セキュアプログラミングに関する最新の書籍です。著者はCERT（コンピュータ緊急対応センター）に在籍しています。

—— Yusuke Sato

HACK #43 -ftrapv で整数演算のオーバーフローを検出する

GCCの-ftrapvオプションを使用すると、符号付き整数同士の加減乗算における整数オーバーフローをランタイムに検出することができます。

本 Hack では、整数オーバーフローを自動的に検出する GCC の機能、-ftrapv オプションを紹介します。このオプションは、GCC のバージョン 3.4 以降で使用することができます。

整数オーバーフローとは

整数同士の四則演算の結果が、その整数型で表現可能な値の上限または下限を飛び越えてしまうことを「整数オーバーフロー」と呼びます。たとえば、次のようなatoi(3)もどきの関数を自作したとしましょう。

```
//
// my_atoi.c
//
int my_atoi(const char *s) {
  int ret = 0;
  assert(s != NULL);
  while(*s != '\0' && isdigit(*s)) {
    const int dig = *s - '0';
    ret *= 10;
    ret += dig;
    ++s;
  }
  return ret;
}
```

この関数は引数 s に "123" などの小さい数字を与えた場合は正常に動作しますが、例えば、Linux/x86 などの ILP32 環境では、"2147483647" を超える数字を与えると誤動作します。"4294967297" を与えると、なんと 1 が戻ってしまいます。

このような予期しない値が戻ることが原因で、（その値を使った別の処理で）バッファオーバーフローなどの致命的な問題が発生することがよくあります。たとえば、HTTPのContent-Lengthヘッダ値の解析に上記の不完全なatoi関数を使用すると、セキュリティ上の問題を引

き起こすおそれがあるでしょう。

GCCによる整数オーバーフローの自動検出

使い方

GCCのコンパイルオプションを工夫すると、符号付き整数同士の加算、減算、乗算におけるオーバーフローを、ランタイムに検出することができます。オーバーフローをチェックしたいプログラムを、-ftrapv と -g オプション付きでコンパイルしてください。

```
% gcc -ftrapv -g -o my_atoi my_atoi.c
% ./my_atoi 4294967297
Aborted
```

オーバーフローを検出すると、プログラムが abort します。

オーバーフロー箇所の特定

-ftrapv オプション付きでコンパイルされたプログラムは、オーバーフローを検出するとSIGABRTをraiseします。ですから、プログラムをgdb上で実行し、SIGABRTで停止した段階でbacktraceを表示すれば、ソースコードのどこでオーバーフローが発生したのかを把握することができます。

```
Program received signal SIGABRT, Aborted.
0x003477a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(gdb) bt
#0  0x003477a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#1  0x003877d5 in raise () from /lib/tls/libc.so.6
#2  0x00389149 in abort () from /lib/tls/libc.so.6
#3  0x08048673 in __addvsi3 ()
#4  0x0804841a in my_atoi (s=0x8048863 "7") at my_atoi.c:9
...
```

仕組み

-ftrapv オプションが使用されると、符号付き整数同士の加算、減算、乗算が、CPU のインストラクションではなく、libgcc.aというGCC付属のライブラリに含まれる関数を用いて行われるようになります。いわゆる soft-float と似たようなイメージです。たとえば、

```
int sqr(int a) {
  return a * a;
}
```

というコードは、次のようにコンパイルされます。

```
pushl   8(%ebp)     # a
pushl   8(%ebp)     # a
call    __mulvsi3   # 掛け算を実行
```

この、__mulvsi3という関数の中で、オーバーフローの有無をチェックしながらの乗算が行われます。

使用上の注意

注意点 1：チェックされない式がある

-ftrapv オプションでチェック可能なのは、符号付き整数同士の演算だけです。

- 符号なし整数同士の演算
- 符号付き整数と符号なし整数の混合演算
- 符号付き整数から符号なし整数への変換（あるいはその逆）
- bit 数の少ない型への代入による切り捨て

などはチェックされません。

特に2番目の混合演算は、整数オーバーフローによるセキュリティホールの原因になることが多いため注意が必要です。C/C++言語仕様の、整数変換の順位（integer conversion rank）、整数拡張（integer promotions）、通常の算術型変換（usual arithmetic conversion）といったルールを頭に叩き込み、GCC に頼らず手作業で美しいコードを書く必要があります。

注意点 2：除算はチェックされない

-ftrapvオプションでチェックされるのは、加算、減算、乗算だけです。除算はチェックされません。これは、Nビット同士の除算の結果が通常はNビットを超えることがないためです。そんな一見安全そうに見える除算なのですが、次の2つの特殊ケースでのみ、問題が発生します。

- 0 による除算
- INT_MIN / -1（環境によっては除算の結果が INT_MAX を超えてしまう）

前者はよく知られていますが、後者はつい忘れがちです。筆者の環境（Linux/x86）で試しに次のプログラムを実行してみたところ、SIGFPE（Floating point exception）でプログラムが異常終了してしまいました。

```
#include <limits.h>
int main() {
    volatile int a = -1;
    printf("%d\n", INT_MIN / a);
    return 0;
}
```

このような除算が行われる可能性のある箇所では、(-ftrapvではチェックされませんので)手動でチェックコードを挿入するようにしてください。

注意点3：古いGCCでは-ftrapvは使えない

GCC 3.3.x以前ではlibgcc.aの実装に問題が見つかっており、-ftrapvが正しく動作しないことが知られています。ご注意ください。

まとめ

GCCの-ftrapvオプションを使用すると、符号付き整数同士の加減乗算における整数オーバーフローをランタイムに検出することができます。これは、ソフトウェアのセキュリティホールの早期発見に役立ちます。

参考文献

- 『Secure Coding in C and C++』(http://www.cert.org/books/secure-coding/)
  "Integer Security" に1つの章を割き、詳細な説明を与えている書籍です。
- 『JIS X 3010:2003　プログラム言語C』(http://www.webstore.jsa.or.jp/)
  ISO C99規格の日本語訳です。第6章に整数演算のルールを掲載しています。

—— Yusuke Sato

HACK #44

Mudflapでバッファオーバーフローを検出する

Mudflapは C/C++ 言語に対応したデバッグ補助機能です。

Mudflapは GCC4で新たに実装されたデバッグ補助機能です。本HackではMudflapの活用方法を紹介します。

Mudflapの概要

Mudflapは、C/C++言語に対応したデバッグ補助機能です。Mudflapを使用すると、次のようなポインタに関連するプログラムの間違いをプログラム実行時に動的に検出することが

できます。

- バッファオーバーフロー
- メモリリーク
- ヌルポインタ参照
- その他、ポインタの誤使用

「**[Hack #54]　Valgrind でメモリリークを検出する**」～「**[Hack #55]　Valgrind でメモリの不正アクセスを検出する**」で紹介する Valgrind に似ていますが、いくつかの違いがあります。

	動作環境	解析対象の再コンパイル・再リンク	解析速度	heap 変数の検査	stack/data/bss 変数の検査
Valgrind (Memcheck)	x86/x86_64/ppc + Linux	不要	比較的低速	可	いまのところ不可
Mudflap	GCC4 が動作する環境	必要	比較的高速	可	可

まず、Mudflap は GCC4 が動作するプラットフォームであれば基本的にどこでも使用できるのが良い点です。著者が試した範囲だけでも、x86/ppc/alpha/sparc と Linux、GCC-4.0.2の組み合わせでMudflapを動作させることができました。また、heap/stack/data/bss上の変数の誤使用をすべて検出できるのもありがたい点です。

一方、まだ完成したばかりのツールということで、出力の読みやすさや実用性といった面では、(著者の主観ですが)まだValgrindに一日の長があるように思います。皆で使って磨いていきましょう。

使い方

Mudflap を利用するためには解析対象のプログラムを再コンパイル・再リンクする必要があります。

コンパイルとリンク

プログラムをコンパイルする際に、-g -fmudflapを付与します。また、リンク時に-lmudflapを付与します。マルチスレッドプログラムの場合は、それぞれ-fmudflapth、-lmudflapthとします。

```
% gcc -g -fmudflap -o testflap testflap.c -lmudflap
```

ここでもし、"mf-runtime.h が見つからない"、あるいは "libmudflap が見つからない" と怒られてしまう場合は、おそらく追加のパッケージが必要です。状況に応じて追加してください。RedHat 系の Linux の場合は、次のコマンドで OK です。

```
# /usr/bin/yum install libmudflap libmudflap-devel
```

実行

出来上がったバイナリを通常通りに実行すると、Mudflapによるチェックが行われます。エラーメッセージは stderr に出力されます。

```
% ./testflap
```

一時的に Mudflap を無効にしたい場合は、環境変数 MUDFLAP_OPTIONS を利用します。

```
% MUDFLAP_OPTIONS="-mode-nop" ./testflap
```

Mudflap による実行速度の低下を最小限に抑えたい場合は、次のオプションが有効です。

```
% MUDFLAP_OPTIONS="-no-timestamps -backtrace=0" ./testflap
```

例

「**[Hack #55] Valgrind でメモリの不正アクセスを検出する**」で、Valgrind では検出できないとしている「stack/bss 上の変数の誤アクセスを」Mudflap に検出させてみましょう。

```
//
// testflap.c
//
static char onbss[128];

int main() {
  char onstack[128] = {0};
  int dummy;

  dummy = onbss[128];   // off-by-one bug
  dummy = onstack[128]; // ditto.

  return 0;
}
```

GCC4.x を使ってコンパイル・リンクします。

```
% gcc -g -fmudflap -o testflap testflap.c -lmudflap
% ./testflap
```

次のような 2 つのエラーが出力されます。無事、Valgrind で検出できなかったバグを Mudflap で検出することができました。

```
mudflap violation 1 (check/read): time=1139169907.370531 ptr=0x80c9b00 size=129
pc=0x3c0332 location=`testflap.c:10 (main)'
      /usr/lib/libmudflap.so.0(__mf_check+0x44) [0x3c0332]
      ./testflap(main+0xc1) [0x8048785]
Nearby object 1: checked region begins 0B into and ends 1B after
mudflap object 0x86451e8: name=`testflap.c:4 onbss'
bounds=[0x80c9b00,0x80c9b7f] size=128 area=static check=3r/0w liveness=3

mudflap violation 2 (check/read): time=1139169907.371680 ptr=0xbfeb77b0 size=129
pc=0x3c0332 location=`testflap.c:11 (main)'
      /usr/lib/libmudflap.so.0(__mf_check+0x44) [0x3c0332]
      ./testflap(main+0x15b) [0x804881f]
Nearby object 1: checked region begins 0B into and ends 1B after
mudflap object 0x8645e38: name=`testflap.c:7 (main) onstack'
bounds=[0xbfeb77b0,0xbfeb782f] size=128 area=stack check=3r/0w liveness=3
```

仕組み

MudflapはGCCと統合されているため、プログラムのどこでポインタアクセスが行われているか、コンパイル時に知ることができます。そこで、Mudflapは検出したポインタアクセスの周辺に、アクセスが正当かチェックするコード(アセンブリ言語で数十命令)を挿入します。チェックには、libmudflap.so に含まれている __mf_check などの関数も利用されます。

また、Mudflapは一部の標準ライブラリ関数の呼び出しをlibmudflap.soに含まれる関数の呼び出しで置換してしまいます。例えば、ソースコード中での memmove 関数の呼び出しは __mfwrap_memmove 関数の呼び出しに自動的に置換されます。

```
% nm testflap2 | grep memmove
U __mfwrap_memmove
```

どのような標準ライブラリ関数が置換されるかは、mf-runtime.hというファイルを探して中を覗いてみればわかります。例えば著者の環境では次のような内容になっていました。redefine_extname という謎の pragma がバイナリアン心をくすぐります。

```
#ifdef _MUDFLAP
  #pragma redefine_extname memcpy __mfwrap_memcpy
  #pragma redefine_extname memmove __mfwrap_memmove
  ...
  #pragma redefine_extname getprotobynumber __mfwrap_getprotobynumber
#endif /* _MUDFLAP */
```

まとめ

Mudflapはポインタ関係のバグを発見するためのツールです。GCC4以降がインストールされていればどのような環境でも使うことができ、またValgrindでは検出できない種類のバグを見つけることが可能です。

参考文献

- 『GCCWiki - Mudflap Pointer Debugging』(http://gcc.gnu.org/wiki/Mudflap%20Pointer%20Debugging)
  Mudflapの公式ページです。
- 『Mudflap: Pointer Use Checking for C/C++ (pdf)』(http://gcc.fyxm.net/summit/2003/mudflap.pdf)
  環境変数MUDFLAP_OPTIONSの詳しい解説があります。

—— Yusuke Sato

HACK #45 -D_FORTIFY_SOURCEでバッファオーバーフローを検出する

GCCの"Automatic Fortification"という機能を利用すると、問題を起こしやすい関数の誤使用を、コンパイル時、あるいはランタイムにチェックすることができます。

C言語には、gets、strcpy、memcpyといった、バッファオーバーフローを起こしやすいとされる関数が多数存在しています。本Hackで紹介する、GCCの"Automatic Fortification(自動要塞化)"という機能を利用すると、こうした関数の誤使用によるバッファオーバーフローをコンパイル時、あるいは実行時に検出することができます。

基本的な使い方

automatic fortificationを使用するためには、ソースコードの再コンパイルが必要です。次のように、-O1以上の最適化をかけ、-D_FORTIFY_SOURCE=1を付与した状態でコンパイルを行ってください。必要な作業はこれだけで、ソースコードの書き換えや特別なライブラリのリンクは必要ありません。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 foo.c
```

このようにコンパイルを行うと、次の2つのタイミングでバッファオーバーフローのチェックが行われます。

チェック時期	チェック内容
コンパイル時	コンパイル時にチェック可能な明らかなバッファオーバーフロー
実行時	それ以外のオーバーフロー

危険な関数の誤使用チェックということで、ユーザから見える機能は、以前よく利用されていた「libsafe」と似ていますが、strcpy などの危険な関数を再実装した共有ライブラリを LD_PRELOAD で滑り込ませる方式の libsafe とは異なり、automatic fortification は、GCC と glibc の連携によって実現されます。

automatic fortification の 2 種類のチェックを、順に見ていきましょう。

コンパイル時のオーバーフローチェック

スタック上に 6 バイトを確保し、7 バイト strcpy するようなコードをコンパイルしてみます。

```
char buf[6];
strcpy(buf, "hello!"); // 終端の '\0' を含めて 7 バイトをコピー
```

すると、次のようなオーバーフローの警告が表示されます。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 foo.c
foo.c:5: warning: call to __builtin___strcpy_chk will always overflow destination
buffer
```

ランタイムのオーバーフローチェック

次に、グローバル変数として6バイトを確保し、そこにコマンドライン引数(argv[1])の値をコピーするコードで試してみます。

```
static char buf[6];
int main(int argc, char **argv) {
  strcpy(buf, argv[1]);
  return 0;
}
```

argv[1]の値はコンパイル時にはわかりませんので、コンパイルは警告なしで成功します。しかし、通常のバイナリとは異なり、bar の実行時に 6 文字以上の引数を与えると、bar がエラーメッセージを残して abort してくれます。

```
% gcc -O1 -D_FORTIFY_SOURCE=1 -o bar bar.c
% ./bar 12345   (5 文字までは問題が起こらない)
% ./bar 123456  (6 文字を与えると ..)
```

```
*** buffer overflow detected ***: ./bar terminated
Aborted
```

ここで、bufをグローバル変数にしたのは特に意味がありません。bufをスタック上に確保してもチェックは行われます。一般に、strcpy、memcpy などのチェック対象関数のコピー先(dest)メモリのサイズをGCCがコンパイル時に把握できるケースでは、ランタイムチェックが有効になります。ですから逆に、次のようなコードではチェックはまったく行われません。

```
// コピー先のメモリサイズが実行時に決まる例
char *buf = malloc(atoi(argv[1]));
strcpy(buf, "hello!");
```

もう1つ、gccではなくg++でコンパイルを行った場合もチェックは一切行われません。ご注意ください。

仕組み

automatic fortification対応のGCCは、__builtin_object_sizeというビルトイン関数を提供します。これは、変数のサイズ(配列長)などをコンパイル時に調べる関数です。コンパイル時にサイズが求められない場合は、このビルトイン関数は-1を戻します。

一方のglibcは、-D_FORTIFY_SOURCEされている場合にかぎり(例えば)getsの定義を次のように変更します。__bos は、__builtin_object_size のことだと思ってください。

```
(/usr/include/bits/stdio2.h より抜粋)
#define gets(__str) \
  ((__bos (__str) == (size_t) -1)                    \
   ? (gets) (__str) : __gets_chk (__str, __bos (__str)))
```

これにより、gets関数に渡された引数__strのサイズがコンパイル時にわかる場合には、本来の gets 関数ではなく __gets_chk という関数が呼ばれるようになります。__gets_chk 関数は、/lib/libc.so.6 に含まれています。__gets_chk の第2引数にはコンパイル時に判明した__strのサイズが渡りますので、__gets_chkはその内部でバッファオーバーフローが起きるかどうかの判断を行うことができるわけです。

チェックの強化

コンパイル時の-D_FORTIFY_SOURCE=1を-D_FORTIFY_SOURCE=2に増やすと、チェックがより厳しくなります。よくわかる変化は、いわゆるformat-string bugがランタイムに検出されるようになることでしょう。printf、vfprintf、syslogなどの関数が、"%n"を含むフォーマット文字列を引数にとって呼ばれると、次のようにプロセスがabortするようになります。

```
% cat baz.c
#include <stdio.h>
int main(int argc, char **argv) {
  int a;
  printf(argv[1]);  // argv[1]に %n が含まれていると abort する
  printf("%n", &a); // 特例: 文字列リテラルに %n が含まれるのは問題ない
  return 0;
}
% gcc -O1 -D_FORTIFY_SOURCE=2 -o baz baz.c
% ./baz %n
*** %n in writable segment detected ***
Aborted
```

書式制御文字%nは、攻撃以外の目的で使用されることがほとんどないため、このチェックはセキュリティを向上させます。

チェックされる関数一覧

チェックされる関数のざっくりとした一覧は、例えば次のコマンドで得ることができます。

```
%grep -r "_chk " /usr/include | sed 's/.*\(__.*_chk\).*/\1/' | sort | uniq
__confstr_chk
__fgets_chk
..
__wmemset_chk
__wprintf_chk
```

筆者の環境では、65 関数ありました。

Mudflap との使いわけ

automatic fortification は、**[Hack #44]** で紹介した GCC の Mudflap 機能と似ていると思われるかもしれません。GCC4.xで利用可能になった点や、バッファオーバーフロー検出に使えるという点は似ていますが、Mudflapがデバッグ用の機能であるのに対して、こちらはリリースビルドに使用できるという点が異なります。

この機能を有効にすることによるランタイムのオーバーヘッドはほとんどありません。すでに一部のLinuxディストリビューションでは、含まれるすべてのバイナリ、ライブラリを-D_FORTIFY_SOURCE付きでコンパイルしているようです。皆さんもぜひ、自作のソフトウェアを配布されるときには、この機能を使ってみてください。

まとめ

GCC/glibcのautomatic fortification機能を用いると、gets、strcpy、memcpy、あるいはprintf、

vfprintf、syslogといった問題を起こしやすい関数の誤使用を、コンパイル時、あるいはランタイムにチェックすることができます。この機能はGCCとglibcの連携によって実現されており、チェックのコストはほんのわずかです。したがって、リリースビルドに対して使用することも可能です。

—— Yusuke Sato

HACK #46 -fstack-protectorでスタックを保護する

GCCの-fstack-protectorオプション(SSP)を利用すると、C/C++で書かれたプログラムのバッファオーバーフローを検出することができます。

stack-smashing protector(SSP、別名ProPolice)は、IBMのHiroaki Etoh氏によって開発されたGCCのパッチです。SSPを使うと、C/C++で書かれたプログラムのバッファオーバーフローを検出することができます。

IBMのページでGCC(～3.4.4)向けのパッチが公開されており、広く利用されてきました。最近、RedHatによってGCC4への移植作業が行われたとのことで、GCC 4.1以降ではパッチなしでSSPを利用することができます。

使い方

プログラムを-fstack-protectorオプション付きでコンパイルすると、SSPが有効になります。SSPで検出可能なのは、スタック上に確保されたchar、signed char、unsigned char配列のあふれです。次のコードで試してみましょう。

```
int main(int argc, char **argv) {
  char buf[8];
  if (argc >= 1) {
    char *s = argv[1], *d = buf;
    while(*s != '\0') *d++ = *s++;
  }
  return 0;
}
```

SSPを有効にしてコンパイルを行い、9文字以上の文字列を引数にしてプログラムを実行すると、SSPによってスタックあふれが検出され、プログラムがabortします。

```
% gcc -v
Target: x86_64-redhat-linux
gcc version 4.1.0 20060214 (Red Hat 4.1.0-0.27)
% gcc -fstack-protector -o overf overf.c
% ./overf 012345678
*** stack smashing detected ***: ./overf terminated
Aborted
```

インターネットに接続するアプリケーションなどは、念のため-fstack-protector付きでコンパイルしておくと安心です。なお、SSPと「**[Hack #45] -D_FORTIFY_SOURCEでバッファオーバーフローを検出する**」で紹介した_FORTIFY_SOURCE機能の併用は、特に問題がありません。

仕組み

SSPは、以下を防止または検出します。

1. バッファオーバーフローによるローカル変数(特にポインタ)の改竄
2. バッファオーバーフローによるreturn addressやsaved ebpの改竄

1、2のどちらも、改竄の成功は「攻撃者の送り込んだコードの実行」など、致命的なセキュリティ問題につながります。

改竄防止、検出の仕組みは次の通りです。

スタックレイアウトの調整

SSPは、関数のローカル変数のスタック上での位置を、通常のGCCとは異なるものに変更します†。具体的には図4-1のように、char配列を最も上位のアドレスに配置します。図を見ればわかるように、通常のGCCでは配列bufをあふれさせることによって、関数ポインタfnの値(指し先)を改竄することが可能ですが、-fstack-protectorでコンパイルされた場合はfnの改竄ができません。

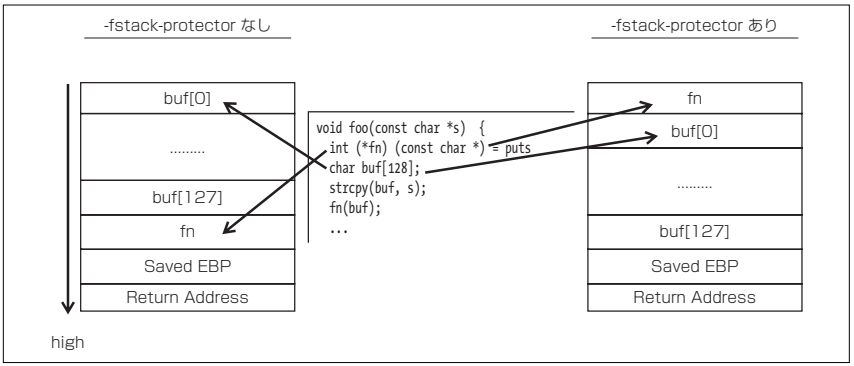


図4-1 SSPの有無によるスタックレイアウトの変化

† 実際には、SSPは関数の引数の位置についてもレイアウトの調整の対象にしますが、紙面の都合でここでは省略します。

このようなスタックレイアウトの調整によって、バッファオーバーフローによるローカル変数の改竄が防止されます。

ガード値(canary)の配置とチェック

さらにSSPは、関数のローカル変数とsaved ebpの間に「ガード」と呼ばれる値を挿入します(図4-2)。

動的ローダによってランダムに決められるこの値は、プログラムを実行するたびに変化するため、事前に予測することはできません。

-fstack-protector付きでコンパイルを行うと、関数fooの冒頭が次のようになります。

```
foo:
        ...(関数fooへの入場処理)...
        movl    %gs:20, %eax        ← ガード値をeaxにロード
        movl    %eax, -8(%ebp)      ← ガード値をスタックに配置
        xorl    %eax, %eax          ← eaxのガード値を消去
        ...(以降、関数foo本体の実行)...
```

また、挿入されたガード値が、バッファオーバーフロー問題によって上書きされていないかどうかのチェックが、関数fooの末尾で行われます。

```
        movl    -8(%ebp), %edx      ← スタック上のガード値をedxにロード
        xorl    %gs:20, %edx        ← オリジナルのガード値と比較
        je      .L5                 ← 両者が一致していれば.L5へ
        call    __stack_chk_fail    ← スタック上のガード値が上書きされている!
                                      __stack_chk_fail関数を呼ぶ
.L5:
        ...(関数fooからの退場処理)...
```

__stack_chk_failは、glibc(またはlibssp)に含まれる関数で、*** stack smashing detected ***というメッセージを表示してプログラムをabortします。バッファオーバーフローによる"return address"や"saved ebp"の改竄が検出できていることになります。

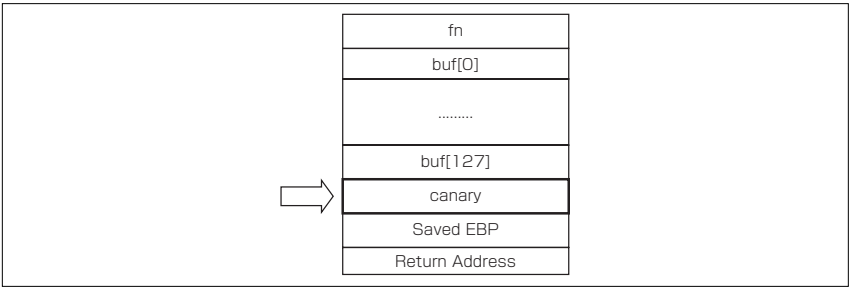


図4-2 ガード値の挿入

動作の調整

SSPは、デフォルトではスタック上に8バイト以上のchar、signed char、unsigned char配列が確保される関数と、allocaを呼んでいる関数だけを保護します。この「8バイト以上」という挙動を「Nバイト以上」に調整したい場合は、GCCの--paramオプションを使用します。実行速度とのトレードオフはありますが、Nの値を小さくするほど保護対象の関数が増えます。

```
% gcc -fstack-protector --param ssp-buffer-size=N foo.c
```

また、無条件にすべての関数を保護したい場合は、以下のようにすればOKです。

```
% gcc -fstack-protector-all foo.c
```

まとめ

GCCの-fstack-protectorオプション(SSP)を利用すると、スタック上の配列をオーバーフローさせる攻撃を受けたことをランタイムに検出し、攻撃によってプログラムが不正動作を起こす前に、プログラムをabortすることができます。

ここからは余談ですが、GCC4.1では、shでもSSPが利用可能とのことです。組み込み用途にGCCを利用している方も、SSPを試してみると良いのではないでしょうか。

参考文献

- 『GCC extension for protecting applications from stack-smashing attacks』(http://www.trl.ibm.com/projects/security/ssp/)
  SSPパッチ(GCC ～ 3.4.4向け)が公開されています。

—— Yusuke Sato

HACK #47

bitmaskする定数は符号なしにする

本Hackでは、ビット操作演算を行う時に注意すべき点を紹介します。

ビット操作演算の例

次のようなコードを見たとき、どこが問題かすぐわかるでしょうか？

```
unsigned long n = 0;
unsigned char *str;
....
n |= (*str & 0xff) << 24;
```

最後の行で*strがunsigned charでnもunsigned longなので、何の問題もなく*strの8ビット値をnの31-24ビット目に設定できるように見えます。これと同様のコードがlibzlib-ruby(Rubyのzlibバインディング)に含まれていましたが、これが原因で不可思議な動作をするようになっていました。本来問題がないgzipされたファイルに対しても、incorrect crc errorを発生するようになっていたのです(Debian Bug#255442)。

x86などのILP32の環境(int、long、ポインタのすべてが32ビットの環境)ではその通りに実行されるのですが、amd64 (x86_64) のようなLP64の環境(longとポインタが64ビットの環境)では予想外の結果を得ることがあります。*strのMSBが立っている場合、例えば0xffだったりする時にnが0xff000000となることを期待しますが、実際に実行するとnは0xffffffffff000000になってしまいます。つまり意図せず63-32ビット目が立った結果が得られてしまうのです。

演算時の型の昇格

なぜこのような結果になるのでしょうか? Cではデータ型が異なるオペランド間で演算を行なう場合に、これらの型を互換性のある型に変換するようになっています。この場合、&演算のオペランドは*strと0xffで、*strはunsigned charですが、0xffはintですので、Cの整数昇格(integer promotions)のルールに従ってintに昇格することになります。ここで問題なのは*strがunsigned charと符号なしであっても&演算をした結果はintと符号付きになってしまうということです。*strが0xffの時(*str & 0xff)の結果は0x000000ffという符号つきintになってしまいます。これを(*str & 0xff) << 24のように24ビット左にシフトすると、結果は0xff000000という符号付きintになります。

このように、*strが符号なしであっても、それから計算した結果(*str & 0xff) << 24は符号付きになってしまうのが罠なのです。

i386のようなILP32の場合はlongも32ビットなので、そのまま代入されるので問題は生じません。しかしlongが64ビットになるamd64のようなLP64の環境では、これをunsigned longであるnに代入する時に、32ビット符号付きintから、64ビット符号付きlongに符号拡張が発生して0xffffffffff000000になり、それが64ビット符号なしlongとして変数nとbitごとのOR演算を行うことになります。

型がどのように評価されているのかを順番に見ていくと次のようになります。

```
*src  => unsigned char
*src & 0xff => unsigned char & int
            => int & int
            => int
(*src & 0xff) << 24  => int << int
                     => int
n |= (*src & 0xff) << 24; => unsigned long |= int
```

```
=> unsigned long |= long
=> unsigned long |= unsigned long
```

まとめ

では、そもそも期待していたような結果を得るためにはどのようなコードを書けばいいのでしょうか？　ここでの問題はビットマスクするのに使った0xffが符号付きintの定数だったことです。つまりこれを符号なし定数とすれば問題は生じません。

```
unsigned long n = 0;
unsigned char *str;
....
n |= (*str & 0xffUL) << 24;
```

このようにビットマスクに使う定数は、0xffU や 0xffUL のように符号なし数値定数として表現すべきです。

—— Fumitoshi Ukai

HACK #48 大きすぎるシフトに注意

大きなシフト幅は、意図しない結果を招くことがあります。

1 << 32 など、大きなシフト幅には注意が必要です。

例

以下のプログラムは1を32ビット左シフトして結果を表示するものですが、どのような結果になるでしょうか？

```
#include <stdio.h>

int main()
{
  unsigned int w = 32;
  printf("%x\n", 1U << w);
  return 0;
}
```

実際に試してみると、最適化を行うかどうかで異なる結果になります。最適化を行わない場合は 1 となり、最適化を行った場合には 0 になります(x86 環境で、コンパイラは Debian の gcc 4.0.3 です)。

```
% gcc tst.c
% ./a.out
1
% gcc -O2 tst.c
% ./a.out
0
```

1を32ビット左シフトすれば、すべてのビットは追い出されて0になることを期待したいところですが、残念ながらそうなるとはかぎらないわけです。

理由

このようになる理由は、x86のシフト命令の仕様にあります。上記の例の最適化を行わないコンパイル結果では、実際にはsall命令が使われます。しかし、sall命令はシフト幅を下位5ビットしか見ません。下位5ビットということは、つまり0から31しか表すことはできないわけで、32は0と等価と見なされます。その結果、1 << 32は1 << 0とみなされ、1という結果になるのです。

また、最適化を行うと0という結果が出てくるのは、シフト演算がgccによるコンパイル時の定数畳み込みで処理されるためです。その時点のシフト演算では、プロセッサのシフト命令の挙動とは関係なく、6ビット(以上)のシフト幅を扱えるため、0という結果になります。

また、gdbで計算したときもgccの定数畳み込みと同様の挙動になります。これはコンパイルしたコードと違う挙動のこともあるので注意が必要です。

```
(gdb) p 1 << w
$1 = 0
```

Cの仕様では、これらの挙動のどちらも許されています。左オペランドの幅以上のシフト幅を右オペランドとして指定したときの結果は未定義なのです。

用途

欲しい結果は0に決まってるんだから32ビットシフトなんてしないよ、と思う人もいるかも知れません。しかし、時には必要な場合があります。例えば、下位nビットのマスクを作る時などが例としてあげられるでしょう。そのようなマスクは(1 << n) - 1として作ることができる、と考えがちですが、1 << 32が0になるとはかぎらないことを考えると、32 <= n ? 0xffffffff : (1 << n) - 1としなければなりません。

まとめ

Cのシフトオペレータは左オペランドの幅以上の値をシフト幅に指定してはいけません。その結果は未定義であり、意図しない結果を招くことがあります。

—— Akira Tanaka

HACK #49 64ビット環境で0とNULLの違いに気を付ける

本 Hack では、0 と NULL の違いが表れる状況について解説します。

0 と NULL

C では、0 という値をもったポインターが NULL となります。NULL は、通常 0 もしくは ((void *)0)と定義されています。そのようなこともあって、NULL を使うべきところで 0 と書いてしまっているプログラムもあります。通常、そのような書き方をしても問題は生じません。

```
char *p = 0;
if (p != 0) ...
```

この場合、p が char * なので演算(= や!=)を実行する前に、オペランドの型を互換性のある型に変換することによって、int の 0 が char * 型に変換されて NULL になるからです。したがって次のような書き方も問題ありません。

```
if (p) ...
```

この場合、ifはブーリアン値で判断することになっていること、式のブーリアン値は0かどうかで判断することになっていることから、次のように解釈されます。

```
if (p != 0) ....
```

逆の条件である

```
if (!p) ..
```

も

```
if (p == 0)
```

と等価になります。

このようにほとんどの場合において、0 と NULL は同じように扱ってもよいように見えます。しかし、それが通用しない場合もありえるのです。

0とNULLが違う場合

前述したとおり、NULLは0というintの値ではありません。0という値をもつポインタがNULLです。つまりポインタという型で解釈されることがコンパイラにわかっていないといけません。

i386のようなILP32環境ではintもlongもポインタも32ビットなので、intの0は通常ポインタのNULLと同じです。しかしながら、IA-64のようなLP64環境では、intは32ビットですが、longとポインタは64ビットです。そのため、intの0だけではNULLにならない場合があります。それが顕著に表れるのが可変長引数を使う場合です。

例えば、次のようなコードを例にして解説してみましょう。

```
struct s *foo(const char *name, ...)
{
    va_list va;
    struct s *sp;
    char *p;
    va_start(va, fmt);
    sp = (struct s *) malloc(sizeof(struct s));
    if (!sp) {
       return 0;
    }
    memset(sp, '\0', sizeof(struct s));
    set_name(sp, name);
    while ((p = va_arg(va, char *))) {
        set_item(sp, p);
    }
    return sp;
}
...
   sp = foo("foo", "bar", 0);
...
```

ここでfoo()は可変長引数をとってnameという名前と残りの引数でitemを設定したstruct sを返す関数です。このコードはLP64環境でうまく動きません。

foo("foo", "bar", 0)のようにfoo()を呼び出す時に、引数は次のようにスタックに積まれています。

```
0 という int
"bar" への const char *
"foo" への const char *
```

foo()の実行では、まず第1引数はconst char *nameに使われます。この場合では"foo"へのポインタがnameになります。残りの引数はva_list経由でアクセスします。foo()ではva_listの内容を次の式で取り出しています。

```
p = va_arg(va, char *)
```

つまり、引数をchar *としてとっていくことになります。まず1つ目、つまり第2引数は"bar"へのポインタなので問題ありません。しかし次の2つ目、つまり第3引数ではintしか渡されていないのにchar *をとろうとしています。ILP32ではどちらも32ビット(4バイト)なので問題はないのですが、LP64ではintは32ビット(4バイト)なのに対して、char *はポインタなので64ビット(8バイト)なので問題になります。つまり呼び出し側ではスタックに32ビット(4バイト)の0しかプッシュしていないのに、呼び出される側(foo())では、64ビット(8バイト)の値をとろうとしているのです。4バイトは0ですが、残りの4バイトはスタック上のゴミを読みとります。運良くそれらも0だと問題ありませんが、0以外の値だと予期せぬ値がitemとしてセットされます。場合によってはスタックをどんどん読んでいってセグメンテーションフォールトしてしまうでしょう。

まとめ

これは0とNULLを混同してしまったことによるバグです。次のようにfoo()を呼び出すべきでした。

```
sp = foo("foo", "bar", NULL);
```

このようにすれば、NULLポインタが呼び出し時にスタックに積まれるのでfoo()でva_arg(va, char *)で取り出しても、ちゃんとNULLポインタが取り出されるので、その時点でwhile()を終了させることができます。

このような関数は自分で書くこともありますが、例えばexecl(3)なども同様なので利用する時は、しっかりNULLで終了させるようにして呼びださなければなりません。

```
int execl(const char *path, const char *arg, ...);
```

—— Fumitoshi Ukai

HACK #50

POSIXのスレッドセーフな関数

マルチスレッドのプログラムで使わないほうがよい関数があります。いくつか用意されている安全な代用関数を使いましょう。

本Hackでは、UNIX上でマルチスレッドなプログラムを書くときの作法と、作法違反のプログラムを無理矢理安全にする方法を解説します。

スレッドアンセーフな関数

POSIX thread-unsafe functions

UNIXの規格[1]では、次の85個の関数はスレッドセーフに作成しなくてよいとされています(http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_09.html)。理由は後で説明しますが、ひとまず「マルチスレッドなプログラムを書くときは、これらの関数を使用してはならない」というのを作法として覚えておいてください。特に、getenv、gethostbyname、gmtime、localtime、rand、readdir、strerror、strtokなどはつい使ってしまいがちな関数であり、注意が必要です。

asctime()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
dbm_clearerr()	fcvt()	getopt()	inet_ntoa()	readdir()
dbm_close()	ftw()	getprotobyname()	l64a()	setenv()
dbm_delete()	gcvt()	getprotobynumber()	lgamma()	setgrent()
dbm_error()	getc_unlocked()	getprotoent()	lgammaf()	setkey()
dbm_fetch()	getchar_unlocked()	getpwent()	lgammal()	setpwent()
dbm_firstkey()	getdate()	getpwnam()	localeconv()	setutxent()
dbm_nextkey()	getenv()	getpwuid()	localtime()	strerror()
dbm_open()	getgrent()	getservbyname()	lrand48()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrand48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlerror()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

(このほか、ctermid、tmpnam、wcrtomb、wcsrtombsも、出力用の引数にNULLが渡されたときはスレッドアンセーフです)

上に挙げた89個の関数を除くと、規格で標準化されている関数はすべてスレッドセーフであることが期待されます。規格で標準化されていない関数(例えばzlibやlibpngの関数)のスレッドセーフティについては、各ライブラリのソースコードやドキュメントを確認してください。

作法違反の実害

glibcなど現実世界のlibcでも、上の表にある関数は通常はスレッドセーフに実装されていません。例えば、localtime関数は次のように実装されていることが多いでしょう。

```
struct tm _tmbuf;
struct tm *localtime(const time_t *timer) {
  /* ... timer引数から年月日などを計算 ... */

  _tmbuf.tm_year = XXX;
```

```
    /* ... 計算結果を構造体に格納 ... */
    _tmbuf.tm_hour = XXX;
    _tmbuf.tm_min  = XXX;
    _tmbuf.tm_sec  = XXX;

    return &_tmbuf; /* グローバル変数へのポインタを返却!! */
}
```

構造体_tmbufが静的な変数である点に注目してください。静的な変数はスレッド間で共有されてしまいますので、スレッド1とスレッド2が次のような順序でほぼ同時にlocaltime関数を呼び出すと、誤動作が起こります。スレッド1は、現在時刻(now)ではなく、はるか昔の時刻(long_time_ago)をprintfしてしまうことでしょう。

	スレッド1 at foo()	スレッド2 at bar()
時刻t	tm = localtime(now);	
時刻t+1		tm2 = localtime(long_time_ago);
時刻t+2	printf("今日は%d月%d日です\n", tm->tm_mon + 1, ...	

この例での誤動作は、「printfする値がおかしくなる」という軽度のものですが、関数によっては「スレッドがNULLポインタ参照を起こす」など、もっと致命的な誤動作の原因になる場合があります。この種のバグは再現性が低いのが常で、検出と除去が本当に大変です。

安全な代用関数

UNIXの規格では、スレッドアンセーフ関数の代わりに使える関数をいくつか定義しています。筆者の知るかぎり次のものがあります。

```
asctime_r       ctime_r         getgrgid_r      getgrnam_r
getpwnam_r      getpwuid_r      gmtime_r        localtime_r
rand_r          readdir_r       strerror_r      strtok_r
```

これらの関数は、マルチスレッドなプログラムから利用してもまったく問題ありません。例えば、localtime関数を使いたい場所では、localtime_r関数を代わりに使えば良いわけです。ただし、locatime関数とlocaltime_r関数は関数のシグネチャが異なるため、ソースコードの書き換えと再コンパイルが必要になります。規格で標準化されていない関数が、libcの独自拡張として提供される場合もあります。例えばglibcには、gethostbyname_rなどの関数が含まれています。

一部のスレッドアンセーフ関数は、各スレッドが明示的にロック処理を行うことで、安全に使用できる場合があります。例えば、putc_unlocked関数はflockfile関数を併用することで安全に使用できます。このことは本Hackでは詳しく説明しませんので、詳細については

man page などを確認してください。

LD_PRELOAD と TLS による Hack

もし、再コンパイルできないバイナリ配布のソフトウェアがlocaltimeなど危険な関数を呼んでいた場合、どうしたらよいでしょう？

```
% nm -D /opt/path/to/broken_proprietary_software | grep localtime
   U localtime  (... 危険な関数を呼んでいる予感!!)
```

このような場合、「**[Hack#60] LD_PRELOAD で共有ライブラリを差し換える**」で紹介するHackと、「**[Hack#26] TLS（スレッドローカルストレージ）を使う**」で紹介したTLSを併用し、libc の localtime 関数を自作の安全なものにすりかえれば、難を逃れることが可能です。

```
#include <time.h>
#include <stdlib.h>

struct tm *localtime(const time_t *timer) {
  static __thread struct tm tmbuf;
  return localtime_r(timer, &tmbuf);
}
```

このようなソースコードを用意して共有ライブラリ化し、バイナリ配布のソフトウェアの実行時に滑り込ませましょう。このような wrapper を自作する際には、「**[Hack#61] LD_PRELOAD で既存の関数をラップする**」で紹介されている、RTLD_NEXT を使った Hack も参考になります。

```
% gcc -D_REENTRANT -O2 -fPIC -shared -o safe_localtime.so -c safe_localtime.c
% LD_PRELOAD=./safe_localtime.so /opt/path/to/broken_proprietary_software
```

これで、バイナリ配布のソフトウェアを一切変更せずに、タイミングに依存した不安定な動作を取り除くことができました。

Windows の場合

_beginthreadex 関数を用いてスレッドを生成すると、localtime などの関数が自動的にTLSを使うようになります。なんともうらやましい環境です。

まとめ

マルチスレッドのプログラムで決して使ってはいけない関数が89個あります。これらの関数の使用は、単に規格違反のコーディングというだけでなく、実際にタイミングに依存した

不安定な動作を起こすことが多く、厄介なバグの原因になりえます。安全な代用関数がいくつか用意されていますので、そちらを使いましょう。

また、他人の書いたプログラムがスレッドアンセーフな関数を呼んでしまっている場合は、LD_PRELOADとTLSを用いたHackで、危険な関数を安全なものにすりかえることが可能です。

参考文献

- 「The Single UNIX Specification, Version 3, 2004 Edition」（http://www.opengroup.org/onlinepubs/009695399/）
  オンラインで閲覧可能な UNIX の規格です。

—— Yusuke Sato

HACK #51 シグナルハンドラを安全に書く方法

どんなときにも誤動作しないシグナルハンドラを書くのは、とても大変です。本Hackでは3つの罠、落し穴を解説します。

「シグナルをシグナルハンドラで処理する」というのはUNIXプログラミングの「いろは」かもしれませんが、どんなときにも誤動作しないシグナルハンドラを書くのは、実はとても大変です。本 Hack では 3 つの罠、落し穴を解説します。

同期シグナルと非同期シグナル

UNIX の規格では、自プロセスの処理が原因で自プロセスに送られてくるシグナルのことを同期シグナルと呼んでいます。例えば次のシグナルは同期シグナルです。

- 0 による除算を行った場合の SIGFPE
- NULL ポインタ参照を行った場合の SIGSEGV
- abort 関数呼び出しによる SIGABRT
- raise 関数呼び出しによる（任意の）シグナル

一方、kill コマンド、kill システムコール、あるいは pthread_kill 関数によって、他プロセスや他スレッドから送られてくるシグナルのことを非同期シグナルと呼びます。本Hackで取りあげるのは、こちらの“非同期シグナル”です。

落とし穴1：シグナルハンドラから非同期シグナルセーフではない関数を呼び出すとまずい

次のような、20秒間寝て終了するだけの、あまり意味のないプログラムを書いてみました。問題のある関数unsafe_funcを、main関数とシグナルハンドラ関数の両方から呼び出しているのがポイントです。

```
//
// async_signal_unsafe.c
//
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// 問題のある関数
void unsafe_func() {
  struct timeval tv = {20, 0};
  pthread_mutex_lock(&lock);
  select(0, NULL, NULL, NULL, &tv); // sleep 20sec.
  pthread_mutex_unlock(&lock);
}

// シグナルハンドラ
void handler(int signo) {
  unsafe_func();
  _exit(1);
}

int main() {
  // シグナルハンドラの登録処理
  struct sigaction sa = {
    .sa_handler = handler,
    .sa_flags   = 0
  };
  sigemptyset(&sa.sa_mask);
  sigaction(SIGHUP, &sa, NULL);

  // 問題のある関数の呼び出し
  unsafe_func();
  return 0;
}
```

このプログラムを実行し、実行開始から10秒ほど経過した時点で、プロセスにSIGHUPを送ると何が起きるでしょうか?

```
% gcc -D_REENTRANT -o async_signal_unsafe async_signal_unsafe.c -lpthread
% ./async_signal_unsafe &  (バックグラウンドで実行)
% killall -HUP async_signal_unsafe
```

ltraceコマンドで観察するとよくわかりますが、このプログラムはデッドロックを起こし

ます。main 関数から unsafe_func 関数が呼ばれ、その中で select システムコールを実行中（寝ている最中）にシグナルを受信し、シグナルハンドラである handler 関数から呼ばれた unsafe_func 関数の中で、ロック済みの mutex を再びロックしてしまうからです。

```
% ltrace ./async_signal_unsafe
...
pthread_mutex_lock(0x8049918, 0x485ff4, 0xbffffa88, 0x804864f, 1) = 0
select(0, 0, 0, 0, 0xbffff9c0 <unfinished ...>
--- SIGHUP (Hangup) ---
pthread_mutex_lock(0x8049918, 0xb7ff6440, 0x7ab9ab2, 0x4cb515, 0x35cfd4  （ここでデッドロック）
```

async-signal-safe 関数

非同期シグナルのシグナルハンドラから安全に呼べる関数のことを「非同期シグナルセーフ（async-signal-safe）関数」と呼びます。次の表が規格上、非同期シグナルセーフとされている関数の一覧です。また、この表にない関数を呼び出した結果は（お使いの環境の man page やソースコードに特に記載がないかぎりは）未規定とされています。

非同期シグナルのシグナルハンドラから async-signal-safe でない関数を呼ぶと、規格上呼び出してはならないというだけでなく、実際にまずいことが起きるケースが多々あります。先ほどの例では、表にない pthread_mutex_lock/unlock 関数を呼び出してしまったためにデッドロックが発生してしまったというわけです。

_Exit()	fpathconf()	read()	sigset()
_exit()	fstat()	readlink()	sigsuspend()
abort()	fsync()	recv()	sockatmark()
accept()	ftruncate()	recvfrom()	socket()
access()	getegid()	recvmsg()	socketpair()
aio_error()	geteuid()	rename()	stat()
aio_return()	getgid()	rmdir()	symlink()
aio_suspend()	getgroups()	select()	sysconf()
alarm()	getpeername()	sem_post()	tcdrain()
bind()	getpgrp()	send()	tcflow()
cfgetispeed()	getpid()	sendmsg()	tcflush()
cfgetospeed()	getppid()	sendto()	tcgetattr()
cfsetispeed()	getsockname()	setgid()	tcgetpgrp()
cfsetospeed()	getsockopt()	setpgid()	tcsendbreak()
chdir()	getuid()	setsid()	tcsetattr()
chmod()	kill()	setsockopt()	tcsetpgrp()
chown()	link()	setuid()	time()
clock_gettime()	listen()	shutdown()	timer_getoverrun()
close()	lseek()	sigaction()	timer_gettime()
connect()	lstat()	sigaddset()	timer_settime()
creat()	mkdir()	sigdelset()	times()
dup()	mkfifo()	sigemptyset()	umask()
dup2()	open()	sigfillset()	uname()
execle()	pathconf()	sigismember()	unlink()

```
execve()          pause()              sleep()          utime()
fchmod()          pipe()               signal()         wait()
fchown()          poll()               sigpause()       waitpid()
fcntl()           posix_trace_event()  sigpending()     write()
fdatasync()       pselect()            sigprocmask()
fork()            raise()              sigqueue()
```

pthreadの関数や、printf系の関数、malloc関数などはついつい呼び出したくなってしまうことが多いものですが、ぐっと我慢して下さい†。

落とし穴2：シグナルハンドラからの非volatile変数の操作に注意

非同期シグナルのシグナルハンドラからグローバル変数を操作する場合、その変数はvolatile修飾されていなければなりません。そうでない変数を操作した結果は未規定です。実害のある単純な例を見てみましょう。

```
//
// non_volatile.c
//
static int sig = 0;
void handler(int signo) {
  sig = 1;
}

int main() {
  // ... シグナルハンドラの登録処理は略 ...

  while(sig == 0);
  return 0;
}
```

シグナルハンドラを登録したあと、無限ループでSIGHUPの到着を待ち、到着したら終了するというプログラムですが、これを筆者の環境で最適化をかけてコンパイルすると正常に動作しませんでした。kill -HUPしても、プロセスが終了しないのです。理由は、gcc -O2 -Sでアセンブリ言語のリストを眺めてみればわかります。

```
  movl    sig, %ebx     # sigの値をEBXレジスタにコピー
.L8:
```

† ただしこれは、非同期シグナルのハンドラにまつわる微妙な問題をまだご存じない読者に向けた、一般論としての注意喚起です。glibcの構造上問題がないなどの理由で、シグナルハンドラから規格上async-signal-safeではない関数を意図的に呼び出すケースはあります。さらに言うなら、「シグナルハンドラで安全に行える処理の限界に挑戦する」のもBinary Hackの楽しみの1つといえるでしょう。**[Hack #53]**のsigsafeを用いたHackなどもその1つです。

```
testl    %ebx, %ebx    # EBX が 0 なら .L8 へ
jne      .L8           #
```

変数sigの値を、初回だけメモリからレジスタにコピーし、その後はレジスタの値だけを監視してループしています。これでは、シグナルハンドラでの変数sigの更新を検出できるわけがありません。変数sigをvolatile修飾することで、この問題は解決します。

落とし穴3：シグナルハンドラからのsig_atomic_t型ではない変数の操作に注意

x86でuint64_tなどの64bit変数を使うと、変数への代入処理は2つのマシン語に分割されます。あるいは逆に、32/64bit変数の操作はアトミックでも、8/16bitの変数への代入は複数命令を必要としてしまうCPUもあります。こういうケースで、もし代入処理の途中でシグナルハンドラに移動してしまうと、ハンドラでの変数の値はまったく予期しないものになってしまいます。ですから、ハンドラから操作するグローバル変数は、マシン語1命令で処理できる型であるほうが無難です。このことを保証するために、signal.hでtypedefされたsig_atomic_tという型を使いましょう。

sig_atomic_t型の変数に代入できる値の範囲は、SIG_ATOMIC_MINとSIG_ATOMIC_MAXというマクロで調べることができます。また、ポータブルなコードを書きたい場合、sig_atomic_t型の変数への代入は、0以上、127以下の値にしておくと良いでしょう。

まとめ

デッドロック、リソースリーク、クラッシュなどを起こさない安全なコードを書きたい場合や、ポータビリティの高いコードを書きたい場合、次のことを守りましょう。

- 非同期シグナルのシグナルハンドラ内では、非同期シグナルセーフ関数だけを呼ぶ
- 非同期シグナルのシグナルハンドラ内では、volatile sig_atomic_t型以外のグローバル変数を操作しない

ところで、「こんな厳しい条件じゃ、シグナルハンドラなんて書けないよ!!」というあなた、大丈夫です。「**[Hack #52]　sigwaitで非同期シグナルを同期的に処理する**」や「**[Hack #53] sigsafeでシグナル処理を安全にする**」で解説されているHackを試して下さい。もっともっと簡単に非同期シグナルを扱うことができます。また、同期シグナルのハンドラについては特にこのような制限はありません。

参考文献

● 「The Single UNIX Specification, Version 3, 2004 Edition, 2.4 Signal Concepts」
(http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html)

—— Yusuke Sato

HACK #52 sigwaitで非同期シグナルを同期的に処理する

本Hackでは、sigwaitという関数を使って「シグナルハンドラを使わないで非同期シグナルをハンドルする」方法を説明します。

「**[Hack #51] シグナルハンドラを安全に書く方法**」というHackでは、非同期シグナルのシグナルハンドラを安全に書くのはなかなか難しいということを説明しました。本Hackでは、安全なシグナル処理をもっと簡単に行うための方法を紹介します。sigwaitというUNIXの規格で標準化されている関数を使うやり方で、「**[Hack #53] sigsafeでシグナル処理を安全にする**」とはまた異なった方法です。

アイデア

非同期シグナルのハンドラ記述に制約が多かったのは、以下の2点が一切予測できないことが原因でした。

- いつどのようなタイミングでシグナルが配送されてくるか
- そしてその結果、いつどのようなタイミングで処理がシグナルハンドラに分岐してしまうか

それならば、シグナルハンドラを使わずに、次のようにシグナルを処理したらどうでしょうか。ここでは再びSIGHUPを例にとります。

1. sigaction関数は一切使用しない
2. すべてのスレッドでSIGHUPをマスクする
3. SIGHUPの処理だけを行なう、専用のスレッドを生成する
4. そのスレッドが、シグナルの到着を見張る(到着チェックの瞬間だけSIGHUPのマスクを解除する)

このようにすると、特定のスレッドの特定の位置のコードを実行している最中にしかSIGHUPがプロセスに配送されてこなくなるため、非同期シグナルのシグナルハンドラ記述にあった

ような制約はなくなります。

sigwait 関数

上記の 4. でシグナルの到着を見張る方法はいくつかありますが、sigwait 関数を使うのがベストでしょう。pause関数やsigsuspend関数でも似た処理は記述できますが、安全に使うのは困難とされています[1]。sigwait 関数は、glibc に付属の man page によると次のような機能を持っています。

書式

```
int sigwait(const sigset_t *set, int *sig);
```

説明

> sigwaitはsetで指定されるシグナルのうちいずれか1つが呼び出しスレッドに配送されるまで呼び出しスレッドの実行を停止する。そして受信したシグナルの数†をsigで指し示される領域に格納して返る。setで指定されるシグナルは sigwait に入るときにブロックされていなければならず、無視されてはならない。

この説明だけではピンとこない場合、POSIX規格を読むと良いでしょう。いくつかのLinuxでは、

```
% man 3p sigwait
```

とすることで、POSIX での sigwait 関数の定義を閲覧することができます。この、セクション 3P のマニュアルは非常に詳細で参考になります。

シグナル処理スレッドの実装例

実際にコードを書いてみましょう。まず、SIGHUPを処理するスレッドのエントリポイントは次のようになります。

```
void *wait_for_sighup(void *dmy) {
  int sig;

  // sigwait 関数は SIGHUP を待つ
  sigset_t ss;
  sigemptyset(&ss);
  sigaddset(&ss, SIGHUP);
```

† 「受信したシグナルの数を」は「受信したシグナルの番号を」の誤記と思われます。

```
    while(1) {
      // SIGHUPの到着を待つ
      if (sigwait(&ss, &sig) == 0) {
        // <<シグナルハンドラ相当の処理>>
        printf("Hello async-signal-safe world!\n");
      }
    }

    return NULL;
  }
```

<<シグナルハンドラ相当の処理>>と書いた場所で、非同期シグナルセーフではない関数を呼び出してもまったく問題ないというのが、本Hack最大のポイントです。非同期シグナルのシグナルハンドラとは違って、printf関数だろうと、malloc関数だろうと、pthread_cond_signal関数だろうと、自由に呼び出すことができるわけです。また、スレッドセーフティに注意する必要はありますが、volatile sig_atomic_t型ではない変数の操作を行っても問題がありません。

main関数の実装例

このスレッドを生成する側の処理は次のようになります。wait_for_sighup関数を含む、すべてのスレッドでSIGHUPがマスクされるように、main関数の先頭でsigprocmask関数を呼び出しています。

```
  int main() {
    sigset_t ss;
    pthread_t pt;
    pthread_attr_t atr;

    // SIGHUPをマスクする(SIGHUPが生起されても保留状態にする)
    sigemptyset(&ss);
    sigaddset(&ss, SIGHUP);
    sigprocmask(SIG_BLOCK, &ss, NULL);

    // SIGHUPを処理する専用スレッドを生成
    pthread_attr_init(&atr);
    pthread_attr_setdetachstate(&atr, PTHREAD_CREATE_DETACHED);
    pthread_create(&pt, &atr, wait_for_sighup, NULL);

    // 以降は通常の処理 ...

    return 0;
  }
```

コンパイルと実行は次のように行います。無事、正常に動作しているようです。

```
% gcc -D_REENTRANT -O2 -o sigwait sigwait.c -lpthread
% ./sigwait &      (バックグラウンドで実行)
[2] 1337
% kill -HUP $!     (そのプロセスにSIGHUPを送る)
Hello async-signal-safe world!
%
```

sigwait 関数の仲間

sigwait 関数には、2 つの variant があります。

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec
timeout);
```

sigaction関数に詳しい方ならすぐにわかると思いますが、sigwaitinfo関数は、どのようなシグナルが送られてきたのかを詳細に知るための構造体siginfo_tを得られるように拡張されています。また、sigtimedwait 関数には、それに加えてシグナルの到着待ち時間を指定する(タイムアウト)機能が付いています。どちらの関数も、sigwait関数とは異なりerrno経由でエラーを通知します。

参考文献

1 『詳解 UNIX プログラミング』(W・リチャード・スティーヴンス著、大木敦雄訳、ピアソン・エデュケーション)

まとめ

本 Hack では、sigwait という関数を用いて「シグナルハンドラを使わないで非同期シグナルをハンドルする」方法を説明しました。この方法では、プロセスにシグナルが配送されたときの処理を、非同期シグナルセーフではない関数を自由に使って記述することができます。そのため、シグナルを安全かつ多彩な方法で処理するのがとても簡単になります。

一般に、シグナルとスレッドを併用したプログラムはテストやデバッグが非常に困難になりがちですが、このsigwaitの例は特別です。厄介な問題を引き起こしにくく、安心して使うことができます。

—— Yusuke Sato

HACK #53 sigsafeでシグナル処理を安全にする

本Hackではsigsafeライブラリを使ってシグナルを簡単確実に処理する方法を紹介します。

sigsafeのインストール

sigsafeはhttp://www.slamb.org/projects/sigsafe/で提供されています。このライブラリはmakeではなく、sconsというツールでビルドプロセスが記述されており、例えば以下のようにしてビルドとインストールを行います。

```
% tar xfz sigsafe-0.1.3.tar.gz
% cd sigsafe-0.1.3
% vi SConstruct        # ファイル冒頭の debug = 1 を debug = 0 とする
% scons
...
% sudo cp src/sigsafe.h /usr/local/include
% sudo cp build-i386-linux-st/libsigsafe.a /usr/local/lib
```

ここでSConstructを編集してdebug = 0としているのは、シグナルを受け取った時に表示されるデバッグメッセージを抑制するためです。

シグナルの難しさ

仮に、以下のようなプログラムを考えましょう。

- 標準入力から標準出力にコピーする(読み込んだデータは必ず出力しなければならない)
- SIGINTを受け取ったら、コピーしたバイト数を標準出力に表示して終了する
- SIGINTを受け取ったら、それ以降は標準入力からの読み込みでブロックしてはならない

このプログラムの骨格を説明します。ここで、読み込んだデータは必ず出力しなければいけないので、readしたデータをwriteしている間はSIGINTを受け付けないようにマスクしてあります。ただし、readでブロックしている最中はSIGINTを受け取る必要があるので、read自体はSIGINTをマスクしていない状態で呼び出しています。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

#define BUFFERSIZE 4096
```

```
volatile long counter = 0;

void handler(int signum)
{
  /* シグナルハンドラの中で何をするか？ */
}

int main(int argc, char **argv)
{
  ssize_t ret;
  size_t wsize;
  char buf[BUFFERSIZE], *p;
  struct sigaction act;
  sigset_t defaultmask, intmask;

  sigemptyset(&intmask);
  sigaddset(&intmask, SIGINT);
  sigprocmask(SIG_SETMASK, &intmask, &defaultmask);

  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, NULL);

  counter = 0;
  for (;;) {
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    /* read 直前にシグナルが来たらどうなるか？ */
    ret = read(0, buf, BUFFERSIZE);
    /* read 直後にシグナルが来たらどうなるか？ */
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == 0)
      return 0;
    if (ret == -1) { perror("read"); exit(1); }
    counter += ret;
    wsize = ret;
    p = buf;
    while (wsize) {
      ret = write(1, p, wsize);
      if (ret == -1) { perror("write"); exit(1); }
      wsize -= ret;
      p += ret;
    }
  }
}
```

ここで問題は、SIGINTのシグナルハンドラで何を行うか、また、readの直前、直後でSIGINTが来た場合にどうするか、というところです。

シグナルハンドラでバイト数を出力して exit する

まず考えられるのが、シグナルハンドラからcounterを出力してexitで終了してしまうというものです。つまり、シグナルハンドラを次のように定義します。

```
void handler(int signum)
{
  printf("%ld\n", counter);
  exit(0);
}
```

しかし、こうしてしまうと、readが返って来た直後、シグナルマスクを設定する前にSIGINTを受け取った時に問題が起きます。その場合、readで読み込んだデータが書き出されず、また、その長さもカウントされません。

シグナルハンドラでフラグをセットする

では、シグナルハンドラの中ではグローバル変数にフラグをセットするだけにするのはどうでしょう？　メインループの中でそのフラグを調べ、セットされていたらカウンタの値を出力して終了するわけです。なお、readでブロックしているときにシグナルを受け取った場合、readが失敗してEINTRとなりますが、そのエラーは無視するようにします。

```
...
volatile int signal_caught = 0;
void handler(int signum)
{
  signal_caught = 1;
}
...
      return 0;
    if (ret == -1 && errno == EINTR) ret = 0;
    if (ret == -1) { perror("read"); exit(1); }
...
    }
    if (signal_caught) {
      printf("%ld\n", counter);
      exit(0);
    }
  }
...
```

しかし、こうすると、readの直前にシグナルが来た時に問題が起こります。シグナルハンドラではフラグをセットするだけなので、シグナルハンドラから返った後にreadシステムコールを呼んでしまうと、readがブロックしてしまうかもしれません。そうするとそのブロックが終わるまで終了処理ができません。ここで標準入力が端末だったとすると、人間が

何か入力するまで終了できないことになってしまいます。

シグナルハンドラから siglongjmp で脱出

シグナルハンドラから普通に出るとブロックしてしまうかも知れないというのであれば、シグナルハンドラの中からsiglongjmpで強制的に外に抜けるのはどうでしょう？　そうすれば、read システムコールの直前にシグナルが来ても、ブロックせずに済みます。

```
...
sigjmp_buf env;
void handler(int signum)
{
  siglongjmp(env, 1);
}
...
  sigset_t defaultmask, intmask;
  if (sigsetjmp(env, 1) != 0) {
    printf("%ld\n", counter);
    exit(0);
  }
  sigemptyset(&intmask);
...
```

しかし、もし、readの直後、シグナルマスクを設定する前にシグナルが来たらどうなるでしょうか。その場合はreadの返り値を失ってしまいます。readの返り値はretに代入されますが、代入が起こる前にシグナルを受け取るかも知れません。その場合、siglongjmp で脱出してしまうと、代入が起きないので返り値がどこにも記録されずに消えてしまいます。readの返り値が得られないと、バイト数を数えられません。当然、読み込んだデータを出力することもできません。これではシグナルハンドラの中からexitするのと同じになってしまいます。

select 近辺で起動したシグナルハンドラから siglongjmp で脱出

それなら、read の前に select して、select の近辺だけでシグナルを受け取るようにして、シグナルハンドラからsiglongjmpで外に出るのはどうでしょう？　selectは入出力が可能かどうかテストするわけですが、これは外部へ影響しないのでその結果を失っても困りません。だからsiglongjmpで脱出してもいいはずだし、selectで入力可能とわかっているときにはreadを呼び出してもブロックしないから、read 近辺ではシグナルを受け付けなくても問題ない、というわけです。

```
...
sigjmp_buf env;
void handler(int signum)
```

```
{
  siglongjmp(env, 1);
}
...
  sigset_t defaultmask, intmask;
  if (sigsetjmp(env, 1) != 0) {
    printf("%ld\n", counter);
    exit(0);
  }
  sigemptyset(&intmask);
...
  for (;;) {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(0, &readfds);
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    ret = select(1, &readfds, NULL, NULL, NULL);
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == -1) { perror("select"); exit(1); }
    ret = read(0, buf, BUFFERSIZE);
    if (ret == 0)
...
```

しかし、selectを使うと、selectとreadの間に他のプログラムがデータを読んでしまったらブロックする、という問題が発生してしまいます。標準入力が端末につながっている場合、端末はいろんなプログラムで共有されていますからあり得ない話ではありません。

なお、select近辺で起動したシグナルハンドラから脱出する方法はsigsetjmp以外にthe pipe trickというものもあります。これは、事前にpipeを作ってその読み込み側をselectで監視しておき、シグナルハンドラの中でpipeの書き込み側に1byte書き込む、というものです。そうすると、シグナルハンドラから戻った後にselectが起動しても、そのpipeから読み込み可能なので即座にselectが終了してブロックしない、というわけです。しかし、このトリックはselectとreadの間に他のプログラムがデータを読んでしまったときの問題は解決しません。

ノンブロッキングI/Oを使う

それならノンブロッキングI/Oにすればいいではないか、と思う人もいるかも知れません。たしかに、ノンブロッキングI/OにすればreadはブロックするかわりにEAGAINエラーになりますので、そのときにはselectからやりなおすことにして、siglongjmpを組み合わせれば、仕様を満たせることになります。

```
...
sigjmp_buf env;
void handler(int signum)
{
```

```
    siglongjmp(env, 1);
  }
  ...
    sigset_t defaultmask, intmask;
    if (sigsetjmp(env, 1) != 0) {
      printf("%ld\n", counter);
      exit(0);
    }
    sigemptyset(&intmask);
  ...
    for (;;) {
      fd_set readfds;
      FD_ZERO(&readfds);
      FD_SET(0, &readfds);
      sigprocmask(SIG_SETMASK, &defaultmask, NULL);
      ret = select(1, &readfds, NULL, NULL, NULL);
      sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
      if (ret == -1) { perror("select"); exit(1); }
      ret = fcntl(0, F_GETFL);
      if (ret == -1) { perror("F_GETFL"); exit(1); }
      ret = fcntl(0, F_SETFL, ret|O_NONBLOCK);
      if (ret == -1) { perror("F_SETFL"); exit(1); }
      ret = read(0, buf, BUFFERSIZE);
      if (ret == 0)
        return 0;
      if (ret == -1 && errno == EAGAIN) continue;
      if (ret == -1) { perror("read"); exit(1); }
  ...
```

しかし、標準入力が端末だとすると、それをノンブロッキングI/Oにするのは迷惑です。端末はさまざまなプログラムで共有されており、それをノンブロッキングI/Oに設定するとその設定もほかのプログラムと共有されます。

実際、ノンブロッキングI/Oは、ほかのプログラムに悪影響を与えることがあります。たとえば、stdioという有名なライブラリは、ノンブロッキングI/Oと相性が悪く、組み合わせるとデータが消失してしまうこともあります(Richard StevensはstdioをノンブロッキングI/Oで使うことを「破滅のレシピ」とまで表現しています)。それに、ノンブロッキングI/Oの設定が複数のプログラムで共有されるということは、このプログラムがノンブロッキングに設定してからreadを呼び出す間に、他のプログラムがブロッキングに戻してしまうかも知れません。そうなると、readはやっぱりブロックしてしまうかも知れず、その直前にシグナルが来てシグナルハンドラがフラグをセットしたとすると、SIGINTが届いたのに標準入力から読もうとしてブロックするという2つ目の案と同じになってしまいます。

sigsafe

シグナルの難しさは、ブロックするシステムコールの直前、直後に来た時の処理にありま

す。システムコールの前に来た時には、sigsetjmpで抜けるのが良いのですが、後に来た時にそうしてしまうとシステムコールの返り値を失ってしまいます。また、システムコールの後に来た時にはそのままreturnで抜ければいいのですが、前に来た時にそうしてしまうとシステムコールでブロックしてしまいます。

なお、システムコールでブロックしている最中にシグナルが来ることも当然あります。その場合カーネルは、いったんユーザレベルに処理を戻してシグナルハンドラを起動しますので、システムコールの前後どちらかに来たのと同じことになります。前後どちらと等価になるかはシステムコールが再起動するかどうかで決まり、再起動するときは前、再起動しない場合は後と等価になります。

というわけで、問題はシステムコールの前か後かをシグナルハンドラから区別できれば解決します。前だったらsiglongjmpし、後だったらそのまま抜ければいいわけです。ところが残念なことにこれを行うポータブルな方法はありません。

しかし、ポータブルでなければ実現できる方法があり、sigsafeは実際に実現しています。その仕掛けは、シグナルハンドラから返り先のアドレス付近のコードを調べ、システムコールを行う割り込み命令の前に返るのか、後に返るのかを確認するというものです。この処理は当然プロセッサごとに異なりますし、OSによっても異なりますが、現在sigsafeでは以下の環境をサポートしています。

- Darwin/ppc (a.k.a OS X)
- FreeBSD/i386
- Linux/alpha
- Linux/i386
- Linux/ia64
- Linux/x86_64
- NetBSD/i386
- Solaris/sparc
- Tru64/alpha

sigsafe による実装

sigsafeを使うと、例としてあげたプログラムは次のように実装できます。ここでは、readのかわりにsigsafe_readという関数を使用しています。この関数はその関数の実行中および実行前にシグナルを受け取った場合、-EINTRを返します。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <errno.h>
#include <sigsafe.h>

#define BUFFERSIZE 4096

long counter = 0;

void handler(int signum, siginfo_t *si, ucontext_t *ctx, intptr_t user_data)
{
  /* SIGINT でプロセスがいきなり終了しないようにシグナルハンドラが必要ですが、中では何もしません */
}

int main(int argc, char **argv)
{
  ssize_t ret;
  size_t wsize;
  char buf[BUFFERSIZE], *p;
  sigset_t defaultmask, intmask;

  sigemptyset(&intmask);
  sigaddset(&intmask, SIGINT);
  sigprocmask(SIG_SETMASK, &intmask, &defaultmask);

  sigsafe_install_handler(SIGINT, handler);
  sigsafe_install_tsd(0, NULL);

  counter = 0;
  for (;;) {
    sigprocmask(SIG_SETMASK, &defaultmask, NULL);
    ret = sigsafe_read(0, buf, BUFFERSIZE);
    sigprocmask(SIG_SETMASK, &intmask, &defaultmask);
    if (ret == -EINTR) {
      printf("%d\n", counter);
      exit(0);
    }
    if (ret == 0)
      return 0;
    if (ret == -1) { perror("read"); exit(1); }
    counter += ret;
    wsize = ret;
    p = buf;
    while (wsize) {
      ret = write(1, p, wsize);
      if (ret == -1) { perror("write"); exit(1); }
      wsize -= ret;
      p += ret;
    }
  }
}
```

コンパイルは、例えば次のように行います。

```
% gcc -I/usr/local/include test.c -L/usr/local/lib -lsigsafe
```

sigsafe_read は関数が起動する前にシグナルを受け取っていた場合、read システムコールは呼び出さず、即座に-EINTRを返します。これを実現するために、sigsafeはどのシグナルを受け取ったかを記録しています(この記録は sigsafe_clear_received でクリアできます)。

このプログラムで sigsafe_read が起動する前に SIGINT を受け取った場合、sigsafe_read が即座に -EINTR を返すため、ブロックせずにバイト数を表示して終了できます。

また、sigsafe_readが終了した後にSIGINTを受け取った場合、sigsafe_readの返り値は実際の read の返り値のままであり、その結果にしたがって読み込んだデータが書き出されます。しかし、sigsafe は SIGINT を受け取ったことを記録しているため、ループの次の繰り返しの sigsafe_read の呼び出しは、即座に -EINTR を返して終了します。その結果、その時点でバイト数を表示して終了します。

つまり、sigsafe_readの直前、直後のどちらでシグナルを受け取っても、正しく処理が行われます。

まとめ

ブロックするシステムコールの直前、直後でシグナルを受け取ったときに、シグナルを遅延せずに処理するのは困難ですが、sigsafe を使うことにより、確実に処理することができます。

—— Akira Tanaka

HACK #54 Valgrind でメモリリークを検出する

Valgrind は、Linux/x86、Linux/amd64、Linux/ppc32 に対応した、プログラムの動作を動的に解析してくれるツールです。

Valgrind(http://valgrind.org/)は、仮想的なCPUの上でプログラムを実行することで、プログラムの動作を動的に解析してくれるツールです。Valgrindを使うと、例えば次のような厄介なバグを効率的に見つけることができます。

- メモリの開放忘れ(memory leak)、メモリの不正な開放、二重開放(double free)
- さまざまな種類の不正なメモリアクセス
- マルチスレッドプログラムでの、メモリアクセスの競合(race condition)

公式ページの説明によると "Valgrind" は「ヴァルグリンド」のように発音するそうです。

インストール

Valgrindは、Linux/x86、Linux/amd64、Linux/ppc32の上で動作します。執筆時点での最新版はv3.1.0となっています。残念ながら、CPUシミュレーションを行うツールという性格上、この他のOS、プロセッサは(公式には)サポートされていません。

Debian GNU/Linux、FedoraCore、CentOSなどのディストリビューションでは標準でValgrindのパッケージ(v2.x系列)が用意されています。最新版である3.1.0を使用したい場合は、ソースコードからビルド、インストールしましょう。普通に./configure && make && sudo make installを実行すればOKです。

使い方

Valgrindの使い方は簡単です。

```
% valgrind --leak-check=full --leak-resolution=high --show-reachable=yes
<target_program>
```

上のようにすると、Valgrind上で<target_program>が動作し、バグレポートを出力してくれます。実際にコードを書いて試してみましょう。

```
//
// valgrind_test1.cpp
//
#include <cstdlib>
int* leaky_foo(void) {
  int *a = new int;
  int *b = new int;  // (バグ1) メモリリーク
  return a;
}

static int *c;
int main(int argc, char **argv) {
  c = leaky_foo();   // (バグ2) メモリリーク(プログラムが終了するまでにcをdeleteしていない)
  char *d = (char *)std::malloc(sizeof(char) * 10U);
  delete[] d;        // (バグ3) mallocしたものをdeleteしている(freeが正しい!)
 }
```

デバッグ対象のプログラムをコンパイルする際には、-gまたは-ggdbオプションを使用し、最適化は最大でも-O1までにします。また、スタティックリンクも避けましょう。これは、mallocなどの関数をvalgrindが置き換えることができないためです。

```
% g++ -g -O0 -o valgrind_test1 valgrind_test1.cpp
% valgrind --leak-check=full --leak-resolution=high --show-reachable=yes ./
valgrind_test1
```

次のようなレポートが報告されます。

```
==31927== Mismatched free() / delete / delete []
==31927==    at 0x400550E: operator delete[](void*) (vg_replace_malloc.c:256)
==31927==    by 0x80486A3: main (valgrind_test1.cpp:16)
==31927==  Address 0x4012098 is 0 bytes inside a block of size 10 alloc'd
==31927==    at 0x400446D: malloc (vg_replace_malloc.c:149)
==31927==    by 0x804868C: main (valgrind_test1.cpp:15)

==31927== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31927==    at 0x40047F8: operator new(unsigned) (vg_replace_malloc.c:164)
==31927==    by 0x80485EB: leaky_foo() (valgrind_test1.cpp:8)
==31927==    by 0x8048639: main (valgrind_test1.cpp:13)

==31927== 4 bytes in 1 blocks are still reachable in loss record 2 of 2
==31927==    at 0x40047F8: operator new(unsigned) (vg_replace_malloc.c:164)
==31927==    by 0x80485DB: leaky_foo() (valgrind_test1.cpp:7)
==31927==    by 0x8048639: main (valgrind_test1.cpp:13)
```

見事にすべての問題を検出してくれました('31927' はプロセス ID です)。

コマンドラインオプション

Valgrindのコマンドラインオプションはたくさんありますが、よく使うのは次の6つです。

- --show-reachable=yesを指定すると、(バグ2)のような無害かもしれないメモリリークも報告されます。
- --trace-children=yesを指定すると、解析中のプログラムがforkで生成したプロセスも解析の対象になります。
- --track-fds=yesを指定すると、ファイルディスクリプタの閉じ忘れを報告してくれるようになります。
- --error-limit=noを指定すると、エラー数が閾値を越えた後でもValgrindが解析を継続してくれます。デバッグ初期にはこのオプションの世話になることがあります。
- --num-callers=<number>で、エラー箇所までのバックトレースを何段まで表示するかを指定できます(デフォルトは12)。C++のプログラムをデバッグする際には指定が必要かもしれません。
- --xml=yesで、出力をXML形式にすることが可能です。出力を自動処理したい場合には便利でしょう。

まとめ

Valgrindは、Linux/x86、Linux/amd64、Linux/ppc32に対応した、プログラムの動作を動

的に解析してくれるツールです。Valgrindを使うと、「メモリの開放忘れ(memory leak)」「メモリの不正な開放、二重開放」などの厄介なバグを効率よく発見することができます。

—— Yusuke Sato

HACK #55 Valgrind でメモリの不正アクセスを検出する

Valgrindを使うと、メモリリークだけではなく、さまざまな種類の不正なメモリアクセスも検出することができます。

「**[Hack #54] Valgrind でメモリリークを検出する**」で紹介したツール "Valgrind" を用いると、メモリリークだけではなく、さまざまな種類の不正なメモリアクセスも検出することができます。本 Hack ではまず、

- 未初期化変数の使用
- 範囲外のメモリアクセス
- 開放済みメモリのアクセス
- コピー元、コピー先の overlap

をValgrindが検出する様子を紹介し、最後にValgrindが検出できない種類のバグを紹介します。

間違ったプログラムの例

次のようなプログラムを書いてみました。このプログラムを普通に実行すると、多くの環境では何も問題なくmain関数が終了しますが、実際にはいろいろとまずい処理を行っています。このプログラムを Valgrind 上で動作させるとどのような結果になるでしょうか。

```
//
// horrible.c
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static char onbss[128];
int main() {
  char onstack[128];
  int uninitialized, dummy;
  char *onheap = (char*)malloc(128);

  dummy = onbss[128];
```

```
  dummy = onstack[150];

  if (uninitialized == 0) {
    printf("hello world!\n");
  }
  close(uninitialized);

  dummy = onheap[128];

  free(onheap);
  dummy = onheap[0];

  strcpy(onstack, "build one to throw away; you will anyway.");
  strcpy(onstack, onstack + 1);

  return 0;
}
```

Valgrind 上での実行

Valgrind上でhorribleコマンドを実行しましょう。メモリリークの検出を行わない場合は、コマンドラインオプションなしで Valgrind を実行できます。

```
% gcc -g -o horrible horrible.c
% valgrind ./horrible
```

すると、5つのエラーが検出されるはずです。順に見ていきましょう。

未初期化変数の使用

スタック上に確保した変数 uninitialized を、初期化しないで使用しています。

```
if (uninitialized == 0) {    // 使用箇所 1
  printf("hello world!\n");
}
close(uninitialized);        // 使用箇所 2
```

Valgrind はこの問題を検出し、次のような 2 つのエラーを表示します。

```
==24754== Conditional jump or move depends on uninitialised value(s)
==24754==    at 0x804848F: main (horrible.c:16)

==24754== Syscall param close(fd) contains uninitialised byte(s)
==24754==    at 0x4187BC: __close_nocancel (in /lib/tls/libc-2.3.4.so)
==24754==    by 0x374E22: __libc_start_main (in /lib/tls/libc-2.3.4.so)
```

範囲外のメモリアクセス

ヒープに128バイトしかメモリを確保していないのに、129バイト目を参照しています。

```
dummy = onheap[128];
```

次のエラーが表示されます。

```
==24754== Invalid read of size 1
==24754==    at 0x80484BB: main (horrible.c:21)
==24754==  Address 0x40120A8 is 0 bytes after a block of size 128 alloc'd
==24754==    at 0x400446D: malloc (vg_replace_malloc.c:149)
==24754==    by 0x8048467: main (horrible.c:11)
```

開放済みメモリのアクセス

メモリをfreeで開放した後にそのメモリを参照しています。

```
free(onheap);
dummy = onheap[0];
```

次のエラーが表示されます。

```
==24754== Invalid read of size 1
==24754==    at 0x80484DB: main (horrible.c:24)
==24754==  Address 0x4012028 is 0 bytes inside a block of size 128 free'd
==24754==    at 0x4004F62: free (vg_replace_malloc.c:235)
==24754==    by 0x80484D1: main (horrible.c:22)
```

コピー元、コピー先のoverlap

strcpy、memcpyなどの関数は、コピー元のメモリ領域とコピー先のメモリ領域が重なっていると誤動作します。

```
strcpy(onstack, "build one to throw away; you will anyway.");
strcpy(onstack, onstack + 1);            // overlapしたコピー
```

次のエラーが表示されます。

```
==24754== Source and destination overlap in strcpy(0xBEEBB980, 0xBEEBB981)
==24754==    at 0x4006047: strcpy (mac_replace_strmem.c:269)
==24754==    by 0x8048511: main (horrible.c:26)
```

Valgrindで検出できないエラー

Valgrindは、スタック上に確保されたメモリや、data/bss領域のメモリの不正なアクセス

は検出しません(http://valgrind.org/docs/manual/faq.html#faq.overruns参照)。そのため、次の2つの間違いは見逃されてしまいました。

```
dummy = onbss[128];   // 範囲外アクセス
dummy = onstack[150]; // 範囲外アクセス
```

これを検出するには、「**[Hack #44] Mudflapでバッファオーバーフローを検出する**」で紹介されている Hack を使う必要があります。

仕組み

valgrindコマンドは、実行開始後すぐに仮想的なCPU(simulated CPU)を生成します。解析対象のプログラムはすべてこの仮想CPU上で実行されるため、Valgrindはプログラムの動作を完全に把握することができます。Valgrindを使用するのに解析対象のプログラムを書き換えたり、再リンクしたりする必要はありませんし、root権限も必要ありません。

仮想CPU上でプログラムを効率的に動作させるため、Valgrindの内部ではx86-to-x86のJIT技術が使用されています。また、解析対象のプログラムがシステムコールを呼ぶ瞬間だけは、仮想CPUの汎用レジスタやスタックポインタの内容を本物のCPUにコピーしてからソフトウェア割り込みをかけます。このあたりが、仕組みとして面白いところでしょう。

詳しい仕組みは、Valgrind 公式ページの「Valgrind Technical Documentation」(http://valgrind.org/docs/manual/mc-tech-docs.html)で解説されています。また、システムコールの仕組みについては「**[Hack #59] システムコールはどのように呼び出されるか**」で解説されていますので、あわせて参照してください。

まとめ

Valgrindを使うと、メモリリークのみならず「範囲外のメモリアクセス」「開放済みメモリのアクセス」といった、メモリ、ポインタ関係のバグも検出することができます。
これは、プログラムの安定性向上やセキュリティ品質向上に非常に役立ちます。

—— Yusuke Sato

HACK #56 Helgrindでマルチスレッドプログラムのバグを検出する

Valgrindは、toolと呼ばれるプラグイン的な仕組みによって拡張することができます。

Valgrindは、toolと呼ばれる(一種のプラグインのような)仕組みによって、さまざまに拡張することができます。標準で次のtoolが用意されています。

Memcheck	メモリリークや不正なメモリアクセスを検出する（デフォルト）
Cachegrind	CPU の L1/L2 キャッシュのヒット率を測定する
Massif	ヒープ使用量を測定する
Helgrind	マルチスレッドプログラムのバグを検出する

自分で新たなtoolを作成することも勿論可能です。実際、Memcheckより厳しいメモリアクセスのチェックを行うツール（Annelid）や、シグナルハンドラの中でまずい処理が行われていないかチェックするツール（Crocus）などの実験的な tool が公開されています（http://valgrind.org/downloads/variants.html）。

Helgrind を試す

ここでは、Helgrindを使ってマルチスレッドプログラムのバグを検出してみましょう。こんなテストプログラムを書いてみました。

```
// エラーチェックは省略しています

// テスト1: ロックなしの変数アクセス
static int count = 1;
void *incr_count(void *p) {
  ++count;
  return 0;
}

// テスト2: 統一されていないロック階層
static pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
void *lock_m1_then_m2(void *p) {
  pthread_mutex_lock(&m1); pthread_mutex_lock(&m2);
  pthread_mutex_unlock(&m2); pthread_mutex_unlock(&m1);
  return 0;
}
void *lock_m2_then_m1(void *p) {
  pthread_mutex_lock(&m2); pthread_mutex_lock(&m1);
  pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);
  return 0;
}

int main() {
  pthread_t t1, t2, t3, t4;
  pthread_create(&t1, NULL, incr_count, NULL);
  pthread_create(&t2, NULL, incr_count, NULL);
  pthread_create(&t3, NULL, lock_m1_then_m2, NULL);
  pthread_create(&t4, NULL, lock_m2_then_m1, NULL);
  pthread_join(t4, NULL); pthread_join(t3, NULL);
  pthread_join(t2, NULL); pthread_join(t1, NULL);
  return count;
}
```

このプログラムには少なくとも2つの問題が潜んでいます。

問題1：ロックなしの変数操作

変数countを、ロックしないで更新しています。

```
void *incr_count(void *p) {
  ++count;
```

このC言語のコードの2行目は、アセンブリ言語では、以下のように複数の命令に分解されますので、

```
movl    count, %eax
incl    %eax
movl    %eax, count
```

複数のスレッドがほぼ同時にincr_count関数を実行すると、変数countが正しく増加しません。

問題2：ロック階層の不統一

ミューテックスのm1、m2を、以下のようにロックしています。

- lock_m1_then_m2関数では、m1→m2の順でロック
- lock_m2_then_m1関数では、m2→m1の順でロック

ロック順序が統一されていません。あるミューテックスをロック中に別のミューテックスをロックする場合、プログラムの全域でそのロック順序を統一するのがセオリーです。これを怠ると(詳細は省きますが)プログラムがデッドロックすることがあります。

使い方

Helgrind上で、先ほどのプログラムを実行してみましょう。コマンドラインオプションで--tool=helgrindを指定すること以外は、Memcheckと同じ使い方です。

```
% gcc -ggdb -o data_race data_race.c -lpthread
% valgrind --tool=helgrind ./data_race
```

ロックなしの変数操作が、次のように検出されました。

```
==4278== Possible data race writing variable at 0x80497D8 (count)
==4278==    at 0x804847F: incr_count (data_race.c:6)
```

```
==4278==    by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278==  Address 0x80497D8 is in data section of /tmp/data_race
==4278==  Previous state: shared RO, no locks
```

ロック階層の不統一も、次のように検出されました。

```
==4278== Mutex 0x80497E0(m1) locked in inconsistent order
==4278==    at 0x1D4B0AA3: pthread_mutex_lock (vg_libpthread.c:1324)
==4278==    by 0x80484FB: lock_m2_then_m1 (data_race.c:22)
==4278==    by 0x1D4AF8D1: thread_wrapper (vg_libpthread.c:867)
==4278==    by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278==  while holding locks 0x80497F8(m2)
==4278==  0x80497F8(m2) last locked at
==4278==    at 0x1D4B0AA3: pthread_mutex_lock (vg_libpthread.c:1324)
==4278==    by 0x80484EB: lock_m2_then_m1 (data_race.c:21)
==4278==    by 0x1D4AF8D1: thread_wrapper (vg_libpthread.c:867)
==4278==    by 0xB000F14F: do__quit (vg_scheduler.c:1872)
==4278==  while depending on locks 0x80497E0(m1)
```

完璧です。

注意点

この Hack の執筆には Valgrind-2.2.0 を使用しました。残念ながら Valgrind-2.4.0 以降では Helgrindが使用できなくなってしまったためです。Valgrind本体の進化にHelgrindが追従できなくなった事が原因のようですが、近いうちに再度サポートされる予定とのことです。首を長くして復活を待ちましょう。

Massif と Cachegrind は 3.1.0 でも使用できます。ぜひ使ってみてください。

まとめ

「**[Hack #54]　Valgrind でメモリリークを検出する**」の Hack で紹介した Valgrind は、tool と呼ばれるプラグイン的な仕組みによって、さまざまに拡張することができます。中でも Valgrindの配布物に標準で含まれているHelgrindというツールは便利で、マルチスレッドプログラムの次のような不具合を検出してくれます。

- 複数スレッドが同一の変数をロックしないで操作している箇所
- mutex のロック階層の不統一

これは、再現しにくい誤動作、再現しにくいデッドロックといった厄介なバグの発見に役立ちます。

—— Yusuke Sato

HACK #57 fakerootで擬似的なroot権限でプロセスを実行する

本Hackでは、擬似的なroot権限をもってプロセスを実行する環境を実現するfakerootについて説明します。

fakerootとは?

Debianなどでパッケージを作成する場合は、ソフトウェアをビルドした後、いきなり現在の環境にmake installするのではなく、make install DESTDIR=$(pwd)/debian/tmpのように、ビルドしているディレクトリ以下のあるディレクトリ$(pwd)/debian/tmpが/であるかのようにして、そこにファイルをインストールしていきます。そして、そのディレクトリを/に見立てたファイルアーカイブを作成することでパッケージを作っていきます(実際にはパッケージの中にはファイルアーカイブの他にコントロール情報やメンテナスクリプトなどが含まれています)。

この時にアーカイブされるファイルのオーナーなどをrootにしたいために、root権限で実行するようになっていました。しかしながら、パッケージを作成するためにroot権限を必要とするのはセキュリティ的にあまりよいことではありません。

そこで開発されたのがfakerootです。fakeroot環境内で実行すると、その中ではまるでrootがファイルを操作しているかのようにそのプロセスには見えるようになります。しかし実際のファイルシステム上では元のユーザ権限でファイル操作をしています。

```
% ls -l
合計 0
-rw-r--r--  1 ukai ukai 0 2006-02-18 01:04 foo
% fakeroot
# id
uid=0(root) gid=0(root) 所属グループ=40(src),1000(ukai)
# ls -l
合計 0
-rw-r--r--  1 root root 0 2006-02-18 01:04 foo
# touch bar
合計 0
-rw-r--r--  1 root root 0 2006-02-18 01:05 bar
-rw-r--r--  1 root root 0 2006-02-18 01:04 foo
# chown www-data:www-data foo
# ls -l
合計 0
-rw-r--r--  1 root     root     0 2006-02-18 01:05 bar
-rw-r--r--  1 www-data www-data 0 2006-02-18 01:04 foo
# exit
% ls -l
合計 0
-rw-r--r--  1 ukai ukai 0 2006-02-18 01:05 bar
-rw-r--r--  1 ukai ukai 0 2006-02-18 01:04 foo
```

このようにfakerootを実行するとfakeroot環境のシェルが起動されます。その中では、擬似的にrootで動いているように見えるのです。id(1)を実行すればrootであると報告されますし、ファイルのオーナーもrootになっています。新しく作ったファイルもオーナーがrootですし、既存のファイルも別のオーナーに変更できるように見えます。

しかし、実際にはこれらは変更されておらずfakeroot環境内だけでそうなっているにすぎません。exitでfakeroot環境から抜けると実際のファイルがどうなっているかがわかります。

Debianでは、パッケージを作る際にroot権限が必要となる作業がありますが、root権限を得るためのコマンドを指定することができるようになっています。sudoなどを使うと本当のroot権限を得て実行することになるわけですが、通常はfakerootを使って擬似的にroot権限を得て実行するようになっています。

fakerootの仕組み

fakerootは単なるシェルスクリプトです。基本的にfakerootコマンドで行なっていることは以下の処理です。

- fakedを起動し、FAKEROOTKEYを得る
- FAKEROOTKEY、LD_LIBRARY_PATH、LD_PRELOADを設定し、コマンド(デフォルトではシェル)を起動する。

fakedは、ファイルのinodeとそれに対応する擬似的なオーナーなどの情報を管理するためのデーモンです。

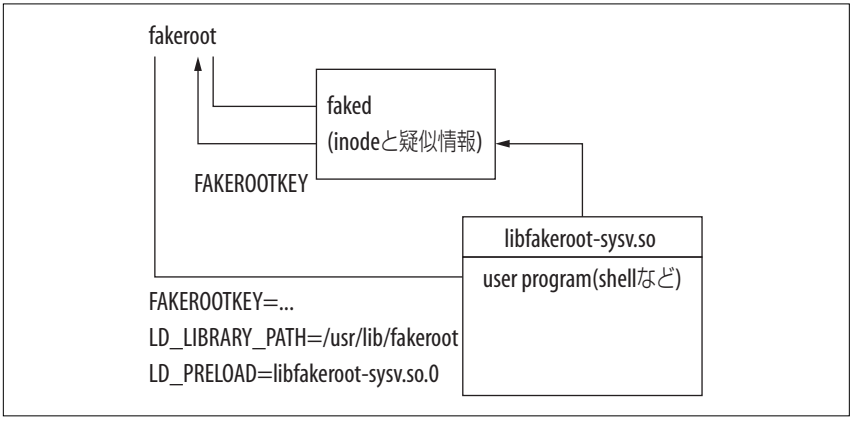
fakerootから実行されるコマンドはLD_LIBRARY_PATHおよびLD_PRELOADで指定される共有オブジェクト(/usr/lib/libfakeroot/libfakeroot-sysv.so.0)でいくつかのAPIがラップされます。

```
LD_LIBRARY_PATH=/usr/lib/libfakeroot LD_PRELOAD=libfakeroot-sysv.so.0
```

libfakeroot-sysv.so.0がラップするのは以下のAPIです。

```
getuid()    geteuid()    getgid()                    getegid()
mknod()
chown()     fchown()     lchown()
chmod()     fchmod()
mkdir()
lstat()     fstat()      stat()(実際には __xlstatなど)
unlink()    remove()     rmdir()                     rename()
```

これらのラップされたAPIは、FAKEROOTKEYの情報を使ってfakedと通信し、擬似的なオーナーなどの情報を得てユーザプログラムに返します。



/usr/lib/libfakeroot-sysv.so.0

実際 faked を通信するためのラッパーをもった共有ライブラリは /usr/lib/libfakeroot/libfakeroot-sysv.so.0ですが、/usr/lib/libfakeroot-sysv.so.0 というものも存在します。これは fakeroot 環境内で suid プログラムをうまく実行するために必要になっています。

fakeroot は LD_LIBRARY_PATH を設定する必要があります。LD_LIBRARY_PATH を設定している場合、LD_PRELOADは絶対パス名を使うことができません。しかしLD_PRELOADが絶対パス名を使っていない場合、バイナリがsuidされていてもLD_LIBRARY_PATHを無視して/libおよび/usr/libからLD_PRELOADに指定された共有ライブラリを見に行ってしまいます。この時、LD_PRELOADで指定した共有ライブラリがなければ、そのsuidバイナリの起動に失敗してしまいます。そのためfakerootでは/usr/lib/libfakeroot-sysv.so.0というダミーのsuidされた共有ライブラリを提供しておくことにより、fakeroot環境下でsuidバイナリを動かそうとした時に、そのダミーの/usr/lib/libfakeroot-sysv.so.0がpreloadされてfakedとは関係なく動くようにしています。

普通のバイナリ

LD_LIBRARY_PATH=/usr/lib/libfakeroot LD_PRELOAD=libfakeroot-sysv.so.0により、/usr/lib/libfakeroot/libfakeroot-sysv.so.0がpreloadされてfakedと通信するようになる。

suid バイナリ

LD_LIBRARY_PATHは無視され、LD_PRELOAD=libfakeroot-sysv.so.0により、suidされた/usr/lib/libfakeroot-sysv.so.0がpreloadされるが、これはダミーなのでpreloadされていない時と同じようにして動く。

まとめ

Debianなどでパッケージを作成する際には、root権限が必要となる時に擬似的なroot権限をもったプロセスを実行するためにfakerootというものが使われています。fakerootは、いくつかのAPIをラップするlibfakeroot-sysv.so.0と、それと通信して擬似的なroot権限の情報を管理するfakedというデーモンからつくられています。

—— Fumitoshi Ukai

5章
ランタイム Hack
Hack #58-86

本章では、スクリプト言語の持つようなランタイム機能をC言語を始めとするコンパイル言語で実現するさまざまなテクニックを紹介します。通常、CおよびC++のプログラムはリフレクションの機能がありませんが、バイナリに残されたシンボルテーブルやデバッグ情報、OSが提供するプロセスや共有ライブラリに関する情報などを駆使することにより、スクリプト言語の持つリフレクションに近い機能を実現することが可能です。

ランタイム Hackは「普通」のプログラムでは実現できないことを実現するHackと考えることができます。本章で紹介するテクニックはスクリプト言語などの処理系や、デバッガ、Java の JIT コンパイラといった高度なプログラムで実際に用いられています。

HACK #58 プログラムが main()にたどりつくまで

Linux において、通常のプログラムが main にたどりつくまでにどのような処理が行われているのかを解説します。

本 Hack では、Linux において通常のプログラムが main にたどりつくまで、どのような処理が行われているのかを説明します。参照しているのはLinux kernel 2.4.27、glibc 2.3.6です。

プログラムの実行開始

hello worldのようなプログラムを作ると、通常次のようなELFバイナリになっています。

```
% file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.2.0, dynamically linked (uses shared libs), not stripped
```

ldd を使うと、libc.so.6 と /lib/ld-linux.so.2 を使うことがわかります。

```
% ldd hello
        libc.so.6 => /lib/libc.so.6 (0x4001c000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

このようなプログラムを、シェル(例えばbash)などから起動する場合を追ってみましょう。まずコマンドラインで"./hello"のように指定されると、シェルがfork(2)して標準入出力のリダイレクト処理をした後に、exec(2)を呼び出すことで、そのプロセスはfork(2)されたbashから指定されたプログラム(この場合は./hello)に置き換わってプログラムが実行されることになります。

execve(2)呼び出し

bashの場合、execute_cmd.cのshell_execve()の中でexecve(2)を使っています。execve(2)はglibcのソースのsysdeps/unix/sysv/linux/execve.cで定義されており、次のようにINLINE_SYSCALLマクロを使ってシステムコールを呼び出しています。

```
return INLINE_SYSCALL (execve, 3, file, argv, envp);
```

このマクロはsysdeps/unix/sysv/linux/i386/sysdep.hで#defineされており、以下のようなアセンブラコードを実行するように展開されます。

```
movl <envp>,%edx
movl <argv>,%ecx
movl <file>,%ebx
movl $11, %eax          # execve
int $0x80
```

このように%eaxにシステムコール番号、%ebx、%ecx、%edxに引数がセットされて、int $0x80でシステムコールを呼び出しています。int命令が実行された時点でソフトウェア割り込みが発生し、ユーザモードからカーネルモードに移行します。

なお、2.6系のカーネルではsysenter/syscall命令を使う場合もあります。**[Hack #59]**を参照してください。

カーネルモードに突入

カーネルは初期化段階にarch/i386/kernel/traps.cのtrap_init()で次のようにシステムコールベクターを初期化しています。

```
set_system_gate(SYSCALL_VECTOR,&system_call); // SYSCALL_VECTOR = 0x80
```

そのために、int $0x80が実行されてソフトウェア割り込みが発生すると、ここで設定したsystem_callに制御が移ります。このsystem_callはarch/i386/kernel/entry.SにあるENTRY(system_call)です。ここに制御が移ってくると、まずレジスタをスタックに保存して、以下の命令により%eaxの内容に従ってsys_call_tableに設定されているアドレスをcallします。

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

sys_call_tableも同じくentry.Sにあり、呼び出し時に設定したのは%eax = 11なので、11番目を見るとsys_execveになっています。

```
ENTRY(sys_call_table)
    (略)
        .long SYMBOL_NAME(sys_unlink)           /* 10 */
        .long SYMBOL_NAME(sys_execve)
        .long SYMBOL_NAME(sys_chdir)
    (略)
```

sys_execveはarch/i386/kernel/process.cのasmlinkage int sys_execve(struct pt_regs regs)にあります。

```
asmlinkage int sys_execve(struct pt_regs regs)
{
      int error;
      char * filename;

      filename = getname((char *) regs.ebx);
      error = PTR_ERR(filename);
      if (IS_ERR(filename))
              goto out;
      error = do_execve(filename, (char **) regs.ecx, (char **) regs.edx, &regs);
      if (error == 0)
              current->ptrace &= ~PT_DTRACE;
      putname(filename);
out:
      return error;
}
```

%ebxに設定されているファイル名をチェックした後、do_execve()を実行しています。システムコールを呼び出した時に%ecxにはargv、%edxにはenvpを設定していることを思い出すと、ここは以下のように呼び出していることがわかります。

```
do_execve(filename, argv, envp, ...)
```

ここまではアーキテクチャ依存なコードで、この後アーキテクチャ非依存なコードになります。

execve(2)の実装 do_execve()

do_execve()はfs/exec.cに定義されています。do_execve()では、まずfilenameで指定されたファイルをファイルシステムからopen_exec()を使って読み込んでいます。ここでパーミッ

ションのチェックなども行います。

openに成功すればstruct linux_binprmの情報を設定していきます。struct linux_binprmには、argvやenvp、uid、gidなどの情報を保持する構造体です。

そしてexecしようとしているファイルの最初のBINPRM_BUF_SIZE（128バイト）を読み込みます。この内容に従って適当なバイナリハンドラを選んで処理を行うことになります。search_binary_handler()でこの読み込んだファイルの先頭に含まれている部分で、マジックナンバーのチェックを行って、struct linux_binfmt *formatsの中からマッチしたバイナリハンドラを探しています。

ELFバイナリの読み込みと実行

ELFに関しては、fs/binfmt_elf.cのelf_formatがバイナリハンドラであり、この中で設定されているload_elf_binaryによってELFヘッダのチェックを行っています。ELFヘッダが正しければkernel_read()を使ってELFのプログラムヘッダテーブルを読み込みます。

プログラムヘッダテーブルの中でPT_INTERPなセグメントを持つプログラムヘッダを見つけて、この内容を読みとります。通常PT_INTERPは.interpセクションになっていて、内容は"/lib/ld-linux.so.2"です。これによりelf_interpreter = "/lib/ld-linux.so.2"となります。

ELFインタープリタの読み込み

elf_interpreterを再びopen_exec()でBINPRM_BUF_SIZE分だけ読み込んでいます。elf_interpreter = "/lib/ld-linux.so.2"はシンボリックリンクをたどると"/lib/ld-2.3.6.so"という共有オブジェクトなので、これを実行することになります。

flush_old_exec()で現在のプログラム（forkしたbash）の情報を消して、新しいプログラム（./hello）のための情報にcurrentの情報を入れ換えていきます。プロセスのメモリマップが初期化されて、プログラムヘッダテーブルでLOADになっているセグメントの部分をメモリ上にマップします。セグメントのfileszよりmemszが大きい場合は、その分のメモリはファイルをマップするかわりに0で埋められるメモリが割り当てられるように設定しています。

最後にelf_interpreterである/lib/ld-2.3.6.soがload_elf_interp()によってメモリ上にマップされます。

その後、currentの情報をさらに設定していって実行を開始します。

実行開始

currentの情報が設定されるとstart_thread()で実行を開始します。

```
start_thread(regs, elf_entry, bprm->p);
```

elf_entryが実行開始アドレスです。ここでelf_entryはelf_interpreterである/lib/ld-2.3.6.soのエントリアドレスになっています。

start_thread()はinclude/asm-i386/processor.hで次のように#defineされています。

```
#define start_thread(regs, new_eip, new_esp) do {              \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));      \
        set_fs(USER_DS);                                       \
        regs->xds = __USER_DS;                                 \
        regs->xes = __USER_DS;                                 \
        regs->xss = __USER_DS;                                 \
        regs->xcs = __USER_CS;                                 \
        regs->eip = new_eip;                                   \
        regs->esp = new_esp;                                   \
} while (0)
```

elf_entryは%eipとして設定しています。その他のレジスタもしかるべき初期値に設定しています。

以上でsearch_binary_handler()、do_execve()、sys_execve()が終了します。

ユーザモードに戻る

sys_execve()を呼び出していたところはarch/i386/kernel/entry.Sでした。

```
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

ここに戻ってきて次のコードが実行されます。

```
        movl %eax,EAX(%esp)             # save the return value
ENTRY(ret_from_sys_call)
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

reschedule()の呼び出しで、kernel/sched.cのasmlinkage void schedule()が呼び出され、その中でinclude/asm-i386/system.hのswitch_to()の呼び出しで、arch/i386/kernel/process.cの__swith_to()が呼び出されます。この中で%esp、%fs、%gsレジスタが入れ替えられてユーザプロセスのコンテクストスイッチが行われます。

コンテクストスイッチの結果、元々execve(2)を呼び出したプロセスに制御が移ってくると、このプロセスの%eipはstart_thread()で設定したようにelf_interpreterのエントリアドレスになっているので、そこから実行を開始することになります。

ELFインタプリタの実行

ELFインタプリタ/lib/ld-2.3.6.soのエントリアドレスは_startになっています。これはglibcのsysdeps/i386/dl-machine.hの#define RTLD_START のコードです。

この中で_dl_start()を呼び出しています。_dl_start()はelf/rtld.cに定義されています。ここではまずbootstrap_map.l_infoを初期化して、sysdeps/i386/dl_machine.hに定義されているelf_machine_load_address()を使って、このELFインタプリタ自体がどのアドレスにあるかを調べます。

次にsysdeps/generic/dl-sysdep.cの_dl_sysdep_start()からelf/rtld.cのdl_main()が呼ばれてきます。dl_main()ではまずprocess_envvars()が呼ばれて環境変数のチェックが行われます。

/lib/ld-2.3.6.soでの環境変数の扱い

以下のような環境変数が/lib/ld-2.3.6.soで使われています。

LD_WARN	警告レベル。出力するかしないか
LD_DEBUG	ダイナミックリンカのデバッグ
LD_VERBOSE	バージョン情報を出力する
LD_PRELOAD	プリロードする共有オブジェクトを指定する
LD_PROFILE	プロファイルをとる共有オブジェクトを指定する
LD_BIND_NOW	遅延バインドをしない
LD_BIND_NOT	ダイナミックリンカではバインドしない
LD_SHOW_AUXV	カーネルから渡らせる補助情報を表示する
LD_HWCAP_MASK	ハードウェアケーパビリティのマスクを設定する
LD_ORIGIN_PATH	バイナリを見つけたパスを設定する
LD_LIBRARY_PATH	共有ライブラリの検索パスを設定する
LD_DEBUG_OUTPUT	デバッグ出力するファイル名を設定する
LD_DYNAMIC_WEAK	weakシンボルも使う
LD_PROFILE_OUTPUT	プロファイル出力するファイル名を設定する
LD_TRACE_PRELINKING	プレリンクをトレースする
LD_TRACE_LOADED_OBJECTS	ロードした共有オブジェクトをトレースする

LD_DEBUGを使うとダイナミックリンカの動作をいろいろと見ることができます。LD_DEBUGでどのようなことができるかはLD_DEBUG=helpで見ることができます。

```
% LD_DEBUG=help /lib/ld-2.3.1.so
Valid options for the LD_DEBUG environment variable are:
```

```
libs        display library search paths
reloc       display relocation processing
files       display progress for input file
symbols     display symbol table processing
bindings    display information about symbol binding
versions    display version dependencies
all         all previous options combined
statistics  display relocation statistics
help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

LD_DEBUG=allとすればすべてが出力されるようになります。またLD_DEBUG_OUTPUTにファイル名を指定すればそこに出力されます。LD_SHOW_AUXVを使うとカーネルから渡された情報を見ることができます。LD_TRACE_PRELINKINGを設定するとどうリンクされるかを確認することができます。LD_TRACE_LOADED_OBJECTSを設定すれば、どの共有オブジェクトをロードするかを確認することができます。つまりlddと同じ動作をします。というよりlddはこの機能で実現されています。

/lib/ld-2.3.6.so の直接実行

/lib/ld-2.3.6.soは直接実行することもできます。

```
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library executables.
This program usually lives in the file `/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file.  This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it.  You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run.  This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.

  --list              list all dependencies and how they are resolved
  --verify            verify that given object really is a dynamically linked
                      object we can handle
  --library-path PATH use given PATH instead of content of the environment
                      variable LD_LIBRARY_PATH
  --inhibit-rpath LIST ignore RUNPATH and RPATH information in object names
                      in LIST
```

例えば--listオプションはlddのような動作をします。

オプションを指定せずにELFバイナリを引数に与えれば、そのELFバイナリを直接実行

する時と同じように実行されます。

共有オブジェクトのロード

まず、プリロードする共有オブジェクトがLD_PRELOADで指定されていれば、その共有オブジェクトをマップします。次に/etc/ld.so.preloadを見て同じくプリロードの処理をします。

プリロードの処理が終われば、実行したELFバイナリのNEEDEDで指定されている共有オブジェクトの処理を行います。

_GLOBAL_OFFSET_TABLE_の再配置

```
ELF_DYNAMIC_RELOCATE (&bootstrap_map, 0, 0);
```

上の部分で、_GLOBAL_OFFSET_TABLE_に対してリロケーションを行っています。ELF_DYNAMIC_RELOCATE()はelf/dynamic-link.hで#defineされています。実際の処理自体はglibcのsysdeps/i386/dl-machine.hのelf_machine_runtime_setup()で行われています。

elf_machine_runtime_setup()では、_GLOBAL_OFFSET_TABLE_の先頭の2エントリの設定をしています。

```
got[1] = (Elf32_Addr) l;  /* Identify this shared object.  */
got[2] = (Elf32_Addr) &_dl_runtime_resolve;
```

この_dl_runtime_resolveというのがシンボルを解決するコードです。共有オブジェクトで定義されている関数を呼び出す時は、この_dl_runtime_resolveを使うことでシンボルの値を解決しています。

プログラム側で関数を呼び出しているところは次のようなコードになっています。

```
.text
      ....
   call    PLT[n]への相対アドレス
      ....
.plt:

    PLT[n]:
       jmp     *GOT[n]
    PLT_resolv[n]:
       push    エントリインデックス
       jmp     レゾルバ

レゾルバ
       pushl    GOT[1]
   jmp    *GOT[2]
```

```
.got:  ...
       GOT[1] この共有オブジェクト自体を識別するための情報
       GOT[2] dl_runtime_resolve
       GOT[3] PLT_resolv[n]
```

最初、コードからPLT[n]を呼び出すとGOT[n]にあるアドレスへジャンプします。遅延レゾルブになっている場合、GOT[n]の初期値は通常PLT_resolv[n]、このjmpの直後になっています。PLT_resolv[n]では、エントリインデックスをpushしてレゾルバのコードへジャンプします。レゾルバはGOT[1]をプッシュして、GOT[2]にあるアドレスにジャンプします。GOT[2]にあるアドレスはダイナミックローダが設定したように_dl_runtime_resolveになっているので_dl_runtime_resolveが呼び出されることになります。_dl_runtime_resolveでは、スタックにあるGOT[1]とエントリインデックスを参照して、GOT[n]をシンボルを解決した値に書きかえます。そしてそこにジャンプして実際の処理を行います。

次にコードから、PLT[n]を呼び出す時もGOT[n]にあるアドレスへジャンプしますが、最初に呼び出した時にGOT[n]にあるアドレスはシンボルが解決されたアドレスになっているので、直接共有オブジェクトにあるコードへジャンプすることができます。

なお、LD_BIND_NOW=1した場合などは、ダイナミックリンカがそのバインド処理を最初に行います。

ELFバイナリのエントリに突入

ダイナミックリンカの処理が終了すると、元々のELFバイナリでエントリアドレスに設定されているアドレスから処理が開始されます。通常は_startというシンボルが付いているコードです。

最終的にglibcのsysdeps/generic/libc-start.cのある__libc_start_main()が実行されて初期設定が行われ、ついにmain()が呼び出されることになります。

```
result = main (argc, argv, __environ);
```

まとめ

ELFバイナリプログラムを実行させた時にどのような処理が行われて実行が開始されるかを説明してきました。一部はカーネルのELFハンドラにより、また一部はELFインタープリタであるダイナミックリンカによりプログラムが動く環境が設定されています。また、ELFインタプリタで利用できる環境変数についても説明しました。

—— Fumitoshi Ukai

HACK #59 システムコールはどのように呼び出されるか

本Hackでは、x86アーキテクチャ上のLinuxにおいて、どのようにシステムコールが呼び出されるのかについて解説します。

現代的なUNIX系のオペレーティングシステムでは、システムの動作モード(コンテキストなどと呼びます)は、ユーザとカーネルという主に2種類(実際は割り込みコンテキストもあります)に分けられます。

ユーザコンテキストは、いわゆる通常のプログラムが走行する動作モードです。これに対して、ハードウェアやシステムを制御する動作モードがカーネルコンテキストです。そして、システムコールはユーザコンテキストからカーネルコンテキストへ処理を切り替え、その際にどの処理をOSに行わせるのか指示します。例えば、端末に文字を出力するシステムコールは「write」ですし、自分のプログラムのプロセスIDを知るシステムコールは「getpid」です。

これらのシステムコールがユーザプログラムから発行されている様子は、straceコマンドで確認できます。詳しくは「**[Hack #82] straceでシステムコールをトレースする**」を参照してください。また、その応用例として「**[Hack #25] glibcを使わないでHello Worldを書く**」を参照してください。

以降では、x86上のLinuxからシステムコールをどのように発行するかの方法を見てみましょう。

syscall関数を使う

システムコールを発行する関数として、syscall()関数があります。例えば、次のプログラムを見てください。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
  int ret;
  ret = syscall(__NR_getpid);
  printf("ret = %d pid = %d\n", ret, getpid());
  return 0;
}
```

ここで、__NR_getpidという名前が使われていますが、これはコンパイル時に「20」という数字に変換されます。名前と番号の対応関係は、/usr/include/bits/syscall.hや、/usr/include/asm-i486/unistd.hなどのファイルに定義されています。実行すると次のように表示さ

れます。

```
% ./syscall-func
ret = 4846 pid = 4846
```

きちんと動いているのがわかります。

int 0x80 を使う

さて、前述のsyscall()関数は、実はglibcの中でシステムコールを呼ぶという操作をラッピングして使いやすくしたものです。では、実際は何が起きているのでしょうか？

古くから使われている方法は、x86プロセッサのソフトウェア割り込み(トラップ)用命令intをユーザプログラムが発行するものです。「int 0x80」命令が発行されると、ユーザプログラムからLinuxカーネルへ制御が移動し、システムコールが発生したことをカーネル側が認識します。それでは、早速例を見てみましょう。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
int main(void)
{
  int ret;
  asm volatile ("int $0x80" : "=a" (ret) : "0" (__NR_getpid));
  printf("ret = %d pid = %d\n", ret, getpid());
  return 0;
}
```

7行目にgccインラインアセンブラが出てきました。簡単に解説すると、int 0x80命令を発行する時に、入力としてeaxレジスタに20を設定し、出力としてeaxレジスタからretへ値を代入しています。

int 0x80命令を発行するときは、eaxレジスタにシステムコールの番号、ここでは20を入れて、getpidを実行しています。なお、引数をとるシステムコールの場合は、引数の順にebx、ecx、edx、eci、edi、ebpレジスタへ値を代入します。

システムコールが完了したら、結果の値がeaxレジスタに入ります。例えばgetpidシステムコールには、結果として現在のプロセスIDが入ります。x86では、エラーが発生していた場合、エラー値(errno)を負数にした値がeaxレジスタに入ってきます。例えば-EINVALが返るような場合は、glibcはerrnoにEINVALを設定し、返り値を-1に書き直します。もし、負数がある範囲より越えれば正常値として扱い、errnoも返り値も書き直しません。アーキテクチャによっては、エラー値を別途持っていることもあります。

sysenterを使う

Linuxカーネル2.6からは、前述のint 0x80以外にも別のシステムコール呼び出し方法であるsysenterがサポートされました。そして、複数のシステムコールをサポートするvsyscallと呼ばれるメカニズムに変更されました。このように変更された最大の理由は、Pentium 4になってからint 0x80命令にかかる時間がPentium IIIよりも遅くなってしまったからです。そこでユーザが何も気にすることなく、カーネル内部でシステムコールの発行メカニズムを変更できるような仕組みが入りました。「**[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**」では触れませんが、Linuxカーネル2.6からはプロセスのメモリ空間の最後に特別な領域が追加されています。/proc/<pid>/mapsの例で言うと、以下の0xffffe000-0xfffff000領域です。

```
% cat /proc/self/maps | grep vdso
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

なお、このアドレスはLinuxカーネルのバージョンによって変化することがあります。以降の例では、0xffffe000からはじまる範囲だったと仮定します。また、プロセス起動ごとにアドレスをランダムに変更することでセキュリティを高める設定になっている場合もあります。ここでは、その設定が無効化されていると仮定します。ここは、実はカーネルが勝手にプロセスへ割り当てた空間で、vsyscall呼び出しをするためのページになっています。vdsoについてはhttp://www.trilithium/johan/2005/08/linux-gate/で詳しく解説されています。このページを読み出してreadelf -Sを実行した結果が次の通りです。1つのELFオブジェクトになっていることが分かります。

```
% dd if=/proc/self/mem of=vdso bs=1 skip=0xffffe000 count=4096
% readelf -S vdso
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .hash             HASH            ffffe0b4 0000b4 000038 04   A  2   0  4
  [ 2] .dynsym           DYNSYM          ffffe0ec 0000ec 000090 10   A  3   5  4
  [ 3] .dynstr           STRTAB          ffffe17c 00017c 000056 00   A  0   0  1
  [ 4] .gnu.version      VERSYM          ffffe1d2 0001d2 000012 02   A  2   0  2
  [ 5] .gnu.version_d    VERDEF          ffffe1e4 0001e4 000038 00   A  3   2  4
  [ 6] .text             PROGBITS        ffffe400 000400 000060 00  AX  0   0 32
...
```

この0xffffe400から始まる領域が、システムコール呼び出しを実際に行います。

```
% objdump -d vdso --start-address=0xffffe400
...
ffffe400 <__kernel_vsyscall>:
ffffe400:       51                      push   %ecx
```

```
ffffe401:       52                      push   %edx
ffffe402:       55                      push   %ebp
ffffe403:       89 e5                   mov    %esp,%ebp
ffffe405:       0f 34                   sysenter
ffffe407:       90                      nop
...
```

実際にサンプルプログラムで確認してみましょう。

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
int main(void)
{
  int pid;
  asm volatile ("call *%2 \n" : "=a" (pid) :
                "0" (__NR_getpid), "S"(0xffffe400));
  printf("ret = %d pid = %d\n", pid, getpid());
  return 0;
}
```

結果は、以下のようになりました。

```
ret = 29261 pid = 29261
```

なお、このアドレス0xffffe400は「**[Hack #27]　glibcでロードするライブラリをシステムに応じて切り替える**」で説明したAT_SYSINFOに登場します。そして、glibcの内部では上記アドレスを直接指定しないで呼び出すようになっています。

まとめ

x86アーキテクチャ上のLinuxにおいて、どのようにシステムコールを呼び出すかの方法について説明しました。そして、内部ではint 0x80命令による方法とsysenterによる方法があることを説明しました。

—— Masanori Goto

HACK #60

LD_PRELOADで共有ライブラリを差し換える

本Hackでは、共有ライブラリの動作を差し替えるためにLD_PRELOADを使う方法を説明します。

共有ライブラリの差し替え

例えばhostname(1)で、ホスト名を実際のホスト名とは違うホスト名を返すように変更するものとします。hostname(1)でホスト名を得るためにはgethostname(3)を使っています。ltrace

してみるとgethostname(3)を呼び出していることがわかります。

```
% hostname
akira.fsij.org
% ltrace hostname
(略)
gethostname("akira.fsij.org", 128)               = 0
(略)
%
```

hostname(1)で、gethostname(3)は共有ライブラリlibc.so.6にあるコードを呼び出しています。

```
% nm -D /bin/hostname | grep gethostname
         U gethostname
% ldd /bin/hostname
        libresolv.so.2 => /lib/tls/i686/cmov/libresolv.so.2 (0xb7fc7000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e91000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb7fea000)
% nm -D /lib/libc.so.6| grep gethostname
000d0e90 W gethostname
```

環境変数LD_PRELOADに共有オブジェクトを指定して実行することで、先にLD_PRELOADの共有オブジェクトがリンクされるようになります。したがって、共有オブジェクトに同名の関数を定義しておくことになり、共有ライブラリよりもLD_PRELOADで指定した共有オブジェクトにあるコードの方がリンクされ、それが呼び出されるようになります。

gethostname(3)を差し替えるためには、例えば次のような共有オブジェクトを作成します。

```
% cat -n gethostname.c
     1  #include <stdlib.h>
     2  #include <string.h>
     3
     4  int
     5  gethostname(char *name, size_t len)
     6  {
     7          char *p = getenv("FAKE_HOSTNAME");
     8          if (p == NULL) {
     9                  p = "localhost";
    10          }
    11          strncpy(name, p, len-1);
    12          name[len-1] = '\0';
    13          return 0;
    14  }
% cc -shared -fPIC -o gethostname.so gethostname.c
%
```

非常に単純なコードですが、環境変数FAKE_HOSTNAMEが設定されていればそれをホスト名と

して、設定されていなければlocalhostをホスト名として返すようなgethostname(3)です。

この共有オブジェクトを使うようにするためには、環境変数LD_PRELOADにそのパス名を指定します。

```
% LD_PRELOAD=./gethostname.so hostname
localhost

% FAKE_HOSTNAME=sai.fsij.org LD_PRELOAD=./gethostname.so hostname
sai.fsij.org
```

カレントディレクトリの時も「./」のように指定する必要があります。「/」を含まない時は、環境変数LD_LIBRARY_PATHで指定されているディレクトリか、標準のライブラリパスにその共有オブジェクトが置かれている必要があります。

```
% LD_PRELOAD=gethostname.so hostname
hostname: error while loading shared libraries: gethostname.so: cannot open shared
object file: No such file or directory

% LD_PRELOAD=gethostname.so LD_LIBRARY_PATH=/tmp hostname
localhost
```

LD_PRELOADで指定する共有オブジェクトに未定義のシンボルがあっても、共有オブジェクトが依存しているライブラリがそれらを提供していればロード時に適宜シンボル値の解決が行われます。

```
% nm -D gethostname.so
00001820 A _DYNAMIC
000018f4 A _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
0000191c A __bss_start
         w __cxa_finalize
         w __gmon_start__
0000191c A _edata
00001920 A _end
000007e0 T _fini
000005d4 T _init
         U getenv
00000734 T gethostname
         U strncpy
% ldd ./gethostname.so
       libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eb8000)
       /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

この場合、getenv(3)およびstrncpy(3)はlibc.so.6にあるものが使われます。

まとめ

LD_PRELOADを使って、共有ライブラリのコードを差し替える方法を説明しました。

—— Fumitoshi Ukai

HACK #61 LD_PRELOADで既存の関数をラップする

本Hackでは、既存の関数をLD_PRELOADでラップする方法を説明します。

既存の関数をラップする

「**[Hack #60] LD_PRELOADで共有ライブラリを差し換える**」ではLD_PRELOADを使って既存の関数を別のコードに置きかえる方法を説明しました。

それでは次のような応用を考えてみましょう。

あるデーモンバイナリがあるのですが、ソケットをbind(2)する時にアドレスを指定せずにINADDR_ANYにしているものがあるとします。普通に使う場合はそれで問題ないのですが、インターフェースが複数あるなどして特定のインターフェースでのみ接続を受け付けるようにしたいと思うことがあります。ソースコードがあってバイナリを作りなおせるのならば、バインドアドレスを設定できるように直すのが正解でしょう。しかしソースがなくてそれができない場合はどうすればいいでしょうか? iptablesなどでカーネルレベルで指定したインターフェースのアドレス以外の接続を切るという方法もありますが、これだとポートが同じでインターフェースごとに別のデーモンを動かす場合に困ってしまいます。

そこで、LD_PRELOADを使って、bind(2)を置きかえることを考えます。例えば次のようなコードになるでしょうか。

```
% cat -n bindwrap0.c
    1  #include <string.h>
    2  #include <sys/types.h>
    3  #include <sys/socket.h>
    4  #include <netinet/in.h>
    5
    6  int
    7  bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)
    8  {
    9      struct sockaddr_in saddr;
   10      if (my_addr->sa_family == AF_INET
   11          && ((struct sockaddr_in *)my_addr)->sin_addr.s_addr == INADDR_ANY) {
   12          struct in_addr sin_addr;
   13          inet_aton(getenv("BIND_ADDR"), &sin_addr);
   14          memset(&saddr, 0, sizeof(saddr));
   15          saddr.sin_family = AF_INET;
   16          saddr.sin_port = ((struct sockaddr_in *)my_addr)->sin_port;
   17          saddr.sin_addr = sin_addr;
```

```
18          return bind(sockfd,
19                      (const struct sockaddr *)&saddr, sizeof(saddr));
20      }
21      return bind(sockfd, my_addr, addrlen);
22  }
```

しかし、これはよく考えるとうまく動作しません。bindの中でまたbindを呼びだしているので再帰ループになってしまうのです。しかも終了条件がないのでどんどん再帰してスタックオーバーフローで死んでしまいます。ではどうすればいいのでしょうか?

既存の関数へのポインタ

要は、この置きかえの bind の中から呼びだす bind が、この新しい bind ではなく元の bind であればいいわけです。元のbindをとるためにはいくつか方法がありますが、最も簡単な方法は RTLD_NEXT ハンドルを使って dlsym(3)で "bind" をとりだすことです(**[Hack #62]** 参照)。RTLD_NEXTハンドルはGNU拡張の特殊なハンドルで、現在の共有オブジェクトの次の共有オブジェクト以降で見つかるシンボルの値をとってきます。この場合、置きかえのbind自体はこの共有オブジェクトに存在するシンボルなので、RTLD_NEXTを使うと以降を探すことになり、libc.so.6にあるbindのシンボルの値をとってくることができるわけです。RTLD_NEXTはGNU拡張なので、dlfcn.hをインクルードする前に_GNU_SOURCEを#defineしておく必要があります。

```
#define _GNU_SOURCE
#include <dlfcn.h>

static int (*bind0)(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
..
bind0 = dlsym(RTLD_NEXT, "bind");
```

その他の方法としては、既存の関数が含まれている共有ライブラリを dlopen(3)して dlsym(3)するというものがあります。dlopen(3)は同じ共有ライブラリを複数ロードしようとしても一度しかマップしないのでこれで特には問題はありません。ラップするものがシステムコールならばsyscall(2)を使ってシステムコールを呼び出すという方法もあります。

ラップする共有オブジェクト

RTLD_NEXT を使えば、ラップする共有オブジェクトのコードは次のようになります。

```
% cat -n bindwrap.c
    1  #define _GNU_SOURCE
    2  #include <dlfcn.h>
    3  #include <string.h>
```

```
 4 #include <sys/types.h>
 5 #include <sys/socket.h>
 6 #include <netinet/in.h>
 7
 8 static int (*bind0)(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
 9 static struct in_addr sin_addr;
10
11 void __attribute__((constructor))
12 init_bind0()
13 {
14     bind0 = dlsym(RTLD_NEXT, "bind");
15     inet_aton(getenv("BIND_ADDR"), &sin_addr);
16 }
17
18 int
19 bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)
20 {
21     struct sockaddr_in saddr;
22     if (my_addr->sa_family == AF_INET
23         && ((struct sockaddr_in *)my_addr)->sin_addr.s_addr == INADDR_ANY) {
24         memset(&saddr, 0, sizeof(saddr));
25         saddr.sin_family = AF_INET;
26         saddr.sin_port = ((struct sockaddr_in *)my_addr)->sin_port;
27         saddr.sin_addr = sin_addr;
28         return (*bind0)(sockfd,
29                         (const struct sockaddr *)&saddr, sizeof(saddr));
30     }
31     return (*bind0)(sockfd, my_addr, addrlen);
32 }
% cc -shared -fPIC -o bindwrap.so bindwrap.c -ldl
```

この例ではinit_bind0()をconstructorという属性にしておくことで、この共有オブジェクトがロードされた時に一度だけbind0とsin_addrの初期化を実行するようにしています。この共有オブジェクトは次のようにして使うことができます。

```
% LD_PRELOAD=./bindwrap.so BIND_ADDR=127.0.0.1 daemon-program
```

このようにbindwrap.soをプリロードするとBIND_ADDR環境変数で指定したアドレスをINADDR_ANYのかわりに使うようになります。netstat -aで見ると0.0.0.0がLocal Addressになっていたのが、この例だと127.0.0.1がLocal Addressになるようになります。

まとめ

既存の関数へのポインタをRTLD_NEXTを使って取得してラップした関数から呼び出すようにするような共有オブジェクトを作ることで、LD_PRELOADして既存の関数をラップすることができるようになります。

—— Fumitoshi Ukai

HACK #62 dlopen で実行時に動的リンクする

本 Hack では、実行時に動的にリンクする dlopen の使い方について説明します。

動的リンク

プログラムによっては、より動的に実行時にどの共有オブジェクトを使うかを決定したい場合があります。例えばブラウザなどのプラグインの場合、ブラウザをビルドする時にはどのプラグインがあるか、そもそもわかっていません。プラグインは後から作成され、実行時にプラグインを動的にリンクするわけです。これはどのように実現しているのでしょうか?

dlopen(3)を使ったプログラムの例

dlopen(3)、dlsym(3)、dlclose(3)を使うプログラムは次のようになります。

```
% cat -n dlsay.c
     1  #include <stdio.h>
     2  #include <dlfcn.h>
     3
     4  int
     5  main(int argc, char *argv[])
     6  {
     7          void *handle;
     8          char *(*msg)();
     9          char *error;
    10
    11          handle = dlopen(argv[1], RTLD_LAZY);
    12          if (handle == NULL) {
    13                  fprintf(stderr, "load error %s: %s\n", argv[1], dlerror());
    14                  exit(1);
    15          }
    16          dlerror(); /* clear error */
    17          msg = (char *(*)()) dlsym(handle, argv[2]);
    18          if ((error = dlerror()) != NULL) {
    19                  fprintf(stderr, "dlsym error %s: %s\n", argv[2], error);
    20                  exit(1);
    21          }
    22          printf("%s\n", (*msg)(argc > 3 ? argv[3] : NULL));
    23          dlclose(handle);
    24          exit(0);
    25  }
    26
% cc -o dlsay dlsay.c -ldl
```

dlsayは最初の引数に共有ライブラリ名を指定し、次の引数で指定したシンボルを文字列を返す関数として、次のコマンドライン引数をその関数の引数として呼び出してそれを表示するプログラムです。これがロードする共有ライブラリは次のようにして作ることができます。

```
% cat -n hello.c
    1  #include <stdio.h>
    2  char *hello(char *arg) {
    3          static char buf[4096];
    4
    5          snprintf(buf, sizeof buf, "hello, %s", arg);
    6          return buf;
    7  }
% cc -shared -fPIC -o hello.o hello.c
```

実行すると次のようになります。

```
% ./dlsay ./hello.so hello world
hello, world
```

hello.so以外にも、char *を引数にとってchar *を返す関数がある共有オブジェクトならどれでも使うことができます。例えば、dlsayは次のように実行することもできます。

```
% ./dlsay libc.so.6 tmpnam
/tmp/fileCZKwXZ
```

dlopen(3)、dlsym(3)、dlclose(3)

dlsayがどのように実現されているか簡単に説明します。

まず、dlopen(3)を使って共有ライブラリをロードします。dlopen(3)は共有ライブラリ名を指定すると、共有ライブラリをロードしそのハンドルを返します。共有ライブラリ名に「/」が含まれている場合、絶対パス名もしくは相対パス名として解釈されます。「./hello.so」の場合はカレントディレクトリのhello.soをロードします。カレントディレクトリにある場合は明示的に「./」を付けるか絶対パス名で指定する必要があります。次のように実行するとエラーになる場合があります(hello.soがどこにあるかに依存します)。

```
% ./dlsay hello.so hello foo
load error hello.so: hello.so: cannot open shared object file: No such file or
directory
```

では、このように「/」が含まれていない場合はどこの共有オブジェクトを使うのでしょうか？「/」が含まれていない場合、バイナリに設定されているRPATHやRUNPATH、もしくは環境変数LD_LIBRARY_PATHなどで指定されたディレクトリを探します。なければ/etc/ld.so.cacheや/lib、/usr/libを探します。「libc.so.6」のように指定した場合、カレントディレクトリがどこであっても「/lib/libc.so.6」をロードします。

dlopen(3)で指定しているRTLD_LAZYというのは、ロード時にシンボルの値の解決をさぼる(lazy)という意味です。RTLD_LAZYの代わりにRTLD_NOWを指定すると、ロード時にシンボルの

値を解決します。もしシンボルの値の解決に失敗すると dlopen(3)自体が失敗となります。RTLD_LAZY の場合は解決できないシンボルがあっても dlsym(3)で調べるまで解決しないので dlopen(3)はエラーになりません。なお、プログラムでは RTLD_LAZY となっていても、環境変数 LD_BIND_NOW に何か文字列を設定しておくと、RTLD_NOW と同じ動作をするようになります。

RTLD_LAZY もしくは RTLD_NOW に対して RTLD_LOCAL や RTLD_GLOBAL のどちらかを設定することができます。RTLD_GLOBAL にすると dlopen(3)で読み込んだ共有オブジェクトにあるシンボルが他の共有オブジェクトでのシンボル解決にも自動的に使われるようになります。デフォルトは RTLD_LOCAL で、dlopen(3)で読み込んだ共有オブジェクトのシンボルは他に影響しません。

dlopen(3)で返されたハンドルを使うことで、その共有オブジェクトに含まれるシンボルの値を得ることができます。そのために使うのが dlsym(3)です。dlsym に共有オブジェクトのハンドルとシンボル名を与えると、その共有オブジェクトの中で定義されているシンボルの値を返します。シンボルが見つからない場合は NULL を返します。エラーの内容は dlerror()で調べられます。例えば次のように実行すると「hi」というのは「hello.so」には存在しないのでエラーになります。

```
% ./dlsay ./hello.so hi world
dlsym error hi: ./dlsay: undefined symbol: hi
```

dlsym(3)で取得できるのはシンボルの値だけなので、そのシンボルがどのような変数、関数なのかは呼びだし側が適当にキャストする必要があります。

dlclose(3)を呼び出すと、そのハンドルに対応した共有オブジェクトのマッピングが外されます。同じ共有オブジェクトが複数 dlopen(3)されている時は、dlopen(3)された回数 dlclose(3)されるまで実際にはマッピングが残ったままになります。

GNU 拡張

glibc で提供されている dlopen(3)にはいくつか GNU 拡張があります。GNU 拡張は _GNU_SOURCE を #define して #include <dlfcn.h> することで使えるようになります。

特殊なハンドル(RTLD_NEXT、RTLD_DEFAULT)

RTLD_NEXT は「次の」共有オブジェクト以降からシンボルを探す時などに使います。共有オブジェクトでなんらかの関数をラップする時などに、ラップする以前の関数へのポインタをとる場合などに RTLD_NEXT を使います。「**[Hack #61] LD_PRELOAD で既存の関数をラップする**」ではこの機能を用いています。RTLD_DEFAULT はグローバルなシンボルを探す時に使います。

```
void *dlvsym(void *handle, char *name, char *version);
```

dlvsym(3)はdlsym(3)と似ていますが、シンボルバージョンを指定してシンボルを探すところが違います。

```
int dladdr(void *addr, Dl_info *info);
```

dladdr(3)はaddrというポインタが含まれるオブジェクトのシンボルの情報を返します。

```
typedef struct {
  const char *dli_fname;    /* 共有オブジェクトのファイル名 */
  void *dli_fbase;         /* 共有オブジェクトのロードされたアドレス */
  const char *dli_sname;    /* シンボル名 */
  void *dli_saddr;         /* シンボルの値 */
} Dl_info;
```

```
int dladdr1(void *addr, Dl_info *info, void **exra_info, int flags);
```

flagにRTLD_DL_SYMENT、RTLD_DL_LINKMAPを指定することで、extra_infoにそれぞれElfNN_Sym *もしくはstruct linkmap *が取得できるようになります。

```
int dlinfo(void *handle, int request, void *arg);
```

共有オブジェクトのさまざまな情報を取得することができます。取得する情報はrequestで指定します。

request	arg	説明
RTLD_DI_LINKMAP	struct linkmap **	リンクマップ情報
RTLD_DI_SERINFO	Dl_serinfo *	ライブラリサーチパス情報
RTLD_DI_SERINFOSIZE	Dl_serinfo *	ライブラリサーチパスの数
RTLD_DI_ORIGIN	char *	$ORIGINのディレクトリ名

その他の注意点

ロードされる共有オブジェクトに_init、_finiというシンボルがあれば、_init()はdlopen(3)時、_fini()はdlclose(3)時に自動的に呼びだされます。

ただし、これらは今となっては推奨されていません。代わりにコンストラクタ、デストラクタを使うべきです。コンストラクタは__attribute__((constructor))属性、デストラクタは__attribute__((destructor))属性を付けて関数を定義しておきます。

まとめ

動的に共有オブジェクトをロードするためのAPI、dlopen(3)について説明しました。

dlopen(3)を使うと実行時にロードすべき共有オブジェクトを変更することができます。RTLD_GLOBAL にしないかぎり dlopen(3)でロードした共有オブジェクトのシンボルは dlsym(3)で取り出さないかぎり他には影響しないので、同じシンボルをもつ複数の共有オブジェクトを同時にロードすることもできます。

—— Fumitoshi Ukai

HACK #63 Cでバックトレースを表示する

本Hackではglibcの関数を使ってCでバックトレース(スタックトレース)を表示する方法を紹介します。

バックトレースとは

バックトレースとは、大ざっぱに言うと、現在の関数に至るまでの道筋です。たとえば、次のRubyプログラムを実行すると、1 / 0の行で例外が発生して、バックトレースの表示とともにプログラムは異常終了します。

```
def foo
  1 / 0
end

def main
  foo
end

main
```

この例では main から foo を呼び、foo の中の 1 / 0 の部分で例外が発生しています。

```
% ruby divide-by-zero.rb
divide-by-zero.rb:2:in `/': divided by 0 (ZeroDivisionError)
        from divide-by-zero.rb:2:in `foo'
        from divide-by-zero.rb:6:in `main'
        from divide-by-zero.rb:9
```

バックトレースは、スタックフレームと呼ばれる一連のデータから復元されます。スタックフレームとは関数呼び出しのたびにスタックに積み上げられる、リターンアドレスや引数のデータをまとめたものです。

同様のプログラムを今度はCで書いて実行してみます。

```
int foo() {
    return 1 / 0;
}
```

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

すると、エラーメッセージは表示されるものの、バックトレースは表示されません。

```
% ./a.out
zsh: 6392 floating point exception (core dumped)  ./a.out
```

Cのプログラムの場合、gdbを使えばバックトレースを表示できます。

```
% gdb a.out core
(gdb) bt
#0  0x08048369 in foo ()
#1  0x08048389 in main ()
```

コンパイル時にgccに-gオプションを付けた場合はファイル名と行番号も表示されます。実行ファイルに埋め込まれたデバッグ情報が用いられるためです。

```
(gdb) bt
#0  0x08048369 in foo () at divide-by-zero.c:2
#1  0x08048389 in main () at divide-by-zero.c:6
```

Cでバックトレースを表示

glibcに含まれるbacktrace()とbacktrace_symbols_fd()を使うと実行中のCプログラムのバックトレースを表示できます。これらの関数の説明はglibcのマニュアルに載っています。以下に簡単な使用例を紹介します。

```
#include <execinfo.h>

void foo() {
    void *trace[128];
    int n = backtrace(trace, sizeof(trace) / sizeof(trace[0]));
    backtrace_symbols_fd(trace, n, 1); //STDOUTへ出力
}

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

このプログラムをgcc -g -rdynamicでコンパイルして実行すると次のようなバックトレースが表示されます。i386上ではbacktrace_symbols_fd()は.dynsymセクション内の情報を利用

する（内部的にdladdrを使っている）ため-rdynamicが必要です。

```
% ./a.out
./a.out(foo+0x1f)[0x8048693]
./a.out(main+0x15)[0x80486d0]
/lib/libc.so.6(__libc_start_main+0xc6)[0x40032e36]
./a.out[0x80485d1]
```

あまり見やすくありませんが、mainからfooが呼ばれていることがわかります。また、GNU binutilsに含まれるaddr2lineを使うとELFバイナリに含まれるデバッグ情報を用いてソースコードのファイル名と行番号を表示できます。

```
% ./a.out | egrep -o '0x[0-9a-f]{7}' | addr2line -f
foo
/home/tmp/c/backtrace.c:5
main
/home/tmp/c/backtrace.c:11
??
??:0
_start
../sysdeps/i386/elf/start.S:105
```

原理の概要とx86での実装

本Hackで紹介したbacktrace関数は、スタックフレームをたどっていくことによって実現されています。関数呼び出しの際、関数の戻る位置を記憶してなければ元の場所に戻ることができませんから、この戻り先はスタックに保存されています。ただしその方法はアーキテクチャによって異なります。例えばx86では、以下のようにしてバックトレースを自力で取得することができます。

```
#include <stdio.h>

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

void print_backtrace() {
    layout *ebp = __builtin_frame_address(0);
    while (ebp) {
        printf("0x%08x\n", ebp->ret);
        ebp = ebp->ebp;
    }
}

void foo() {
    print_backtrace();
}
```

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

x86では通常、ebpレジスタがスタックフレームへのポインタになっていて、これはGCC拡張の__builtin_frame_addressやインラインアセンブラなどで取得できます。このサンプルでは単にアドレスを表示しただけですが、「**[Hack #67] libbfdでシンボルの一覧を取得する**」などと組み合わせれば関数名などの情報も表示することが可能になるでしょう。

x86のスタックフレームについては、glibcのbacktrace.c中のコメントにある下記の図がわかりやすいでしょう。

```
        +-----------------+       +-----------------+
%ebp -> | %ebp last frame--------> | %ebp last frame--->...
        |                 |       |                 |
        | return address  |       | return address  |
        +-----------------+       +-----------------+
```

異常終了時のバックトレース

環境変数LD_PRELOADに/lib/libSegFault.soをセットすると、プログラムの異常終了時にバックトレースを表示できます。

試しに最初の最も単純なCのプログラムをLD_PRELOAD=/lib/libSegFault.so SEGFAULT_SIGNALS=allをセットした上で実行すると次のようなメッセージが表示されました。

```
% export LD_PRELOAD=/lib/libSegFault.so
% export SEGFAULT_SIGNALS=all
% ./a.out
*** Floating point exception
Register dump:

 EAX: 00000001   EBX: 40150880   ECX: 00000001   EDX: 00000000
 ESI: 40016540   EDI: bfffe894   EBP: bfffe828   ESP: bfffe824

 EIP: 080485f9   EFLAGS: 00010286

 CS: 0023   DS: 002b   ES: 002b   FS: 0000   GS: 0000   SS: 002b

 Trap: 00000000   Error: 00000000   OldMask: 00000000
 ESP/signal: bfffe824   CR2: 00000000

Backtrace:
./a.out(foo+0x15)[0x80485f9]
./a.out(main+0x15)[0x8048619]
/lib/libc.so.6(__libc_start_main+0xc6)[0x40036e36]
./a.out[0x8048541]
```

```
Memory map:

08048000-08049000 r-xp 00000000 09:00 5538441    /home/satoru/tmp/a.out
08049000-0804a000 rw-p 00000000 09:00 5538441    /home/satoru/tmp/a.out
40000000-40016000 r-xp 00000000 08:01 700392     /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 08:01 700392     /lib/ld-2.3.2.so
40017000-40018000 rw-p 00000000 00:00 0
40018000-4001b000 r-xp 00000000 08:01 700650     /lib/libSegFault.so
4001b000-4001c000 rw-p 00002000 08:01 700650     /lib/libSegFault.so
40021000-40149000 r-xp 00000000 08:01 700628     /lib/libc-2.3.2.so
40149000-40151000 rw-p 00127000 08:01 700628     /lib/libc-2.3.2.so
40151000-40154000 rw-p 00000000 00:00 0
bfffd000-c0000000 rwxp ffffe000 00:00 0
zsh: 11875 floating point exception (core dumped)  ./a.out
```

このバックトレースでは、0による除算が発生しているfoo()を含めてスタックトレースが表示されています。

/lib/libSegFault.soのラッパーとして、セグメンテーションフォルト(segmentation fault)専用のcatchsegvコマンドがあります。catchsegvはただのシェルスクリプトです。catchsegvを使う場合は以下のように実行します。

```
% catchsegv ./a.out
```

まとめ

glibcの関数を使えば、Cでも簡単にバックトレースを表示できることがわかりました。デバッグ用の情報として使うと便利なのではないかと思います。本Hackのポイントは以下の3つです。

- glibcにはbacktrace()、backtrace_symbols()、backtrace_symbols_fd()が含まれている。
- 環境変数LD_PRELOADに/lib/libSegFault.soを指定するだけで異常終了時にバックトレースを表示できる。
- セグメンテーションフォルトを捕まえるだけならcatchsegvコマンドが使える。

—— Satoru Takabayashi

HACK #64 実行中のプロセスのパス名をチェックする

本Hackでは、実行中のプロセスのパスを調べる方法を紹介します。これはOSや用途に応じてさまざまな方法を使い分ける必要があります。

実行されたプロセスだけでは取得できない情報を調べたい場合、実行ファイルを読み込んでそこから情報を得る必要があります。本Hackでは実行ファイルのパスを取得する方法を

紹介します。これはとても簡単なことに思われますが、標準Cだけで確実に行う方法は存在しないため、OSや用途に応じてさまざまな方法を使い分ける必要があります。

argv[0]からの取得

mainの第2引数であるargvには、実行された時のコマンドライン文字列の配列が格納されています。その最初の要素であるargv[0]を見ることによって、多くの場合、実行ファイルのパスがわかります。しかし、これだけではシェルが環境変数$PATHの中から探索した場合には、ファイルのパスを取得できません。例えば、シェルに対してlsコマンドを発行した場合のargv[0]は"ls"になりますが、普通、実行ファイルのフルパスは/bin/lsなどでしょう。

パスを取得する方法の1つとして、argv[0]からパスを得られればそれを用い、得られなければシェルの挙動を真似てパスを調べる方法があります。この場合、環境変数$PATHで示されるディレクトリを順に探して行って、ファイルが見つかったらそれをパスとします。この方法は多くのOSで共通して使えます(UNIXとWindowsでは環境変数$PATHのセパレータが異なることには注意して下さい)が、実装がいくらか面倒であることや、実行時にコストがいくらかかかることが欠点です。

また、プログラムがUNIX系OSなどでshebang(ファイルの先頭行で#!を用いて自身のインタープリタを指定する宣言)を用いて起動された場合、argv[0]はOS依存になることに注意しましょう。shebangでフルパスが指定された場合、多くのOS(SysVR4、SunOS、Solaris、IRIX、BSDI、BSD-OS、OpenBSD、DU、Unixware、Linux 2.4、FreeBSD)では、そのフルパスがargv[0]として渡されますが、いくつかのOS(Tru64 4.0、AIX 4.3、5.1、Linux 2.2)ではパスではなく実行ファイルの名前だけとなり、HP-UXでは(実行ファイルではなく)スクリプトのフルパスとなります。詳しくは「#! - the Unix truth as far as I know it.」(http://homepages.cwi.nl/~aeb/std/hashexclam.html)に記述されています。

UNIX系OSのprocfsを利用する

OSが実行ファイルのパスを取得する方法を提供している場合、それを利用するのが手軽で安価な解となります。

UNIX系OSのprocfsを利用できれば、それがもっとも手軽な方法の1つとなります。procfsとは、実行中のプロセスの情報をOSが提供するために普通/procにマウントされている、仮想的なファイルシステムです。/procの下にはプロセスIDのディレクトリがあり、その下に個々のプロセスの情報を持つ仮想ファイルがあります。

自分自身のプロセスの情報を調べたい場合、Linuxでは/proc/selfが自分のプロセスIDのディレクトリへのシンボリックリンクとなっているのでその中を調べるのが簡単です。また、各プロセスIDのディレクトリの中にはexeという名前のシンボリックリンクがあり、このシ

ンボリックリンクの先が実行されているファイルになっているため、/proc/self/exe シンボリックリンクの値を調べることによってフルパスを得ることができます。

シンボリックリンクの値を調べるにはreadlink(2)を用います。以下にprocfsを持つLinux環境で実行ファイルのフルパスを表示するプログラムを示します。

```
#include <unistd.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    char fullpath[4096];  // パスの最大長はシステムごとに違うので要注意
    ret = readlink("/proc/self/exe", fullpath, 4096);
    assert(ret != -1);
    printf("%s\n", fullpath);
    return 0;
}
```

procfsはOS依存ですので、"/proc/self/exe"はOSの種類やバージョンによって異なります。例えばFreeBSDでは"/proc/curproc/file"を用いて下さい。

この方法を応用すれば、"/proc/<process_id>/exe"を調べることによって、プロセスIDがわかっている他のプロセスのフルパスを得ることもできます。また、"/proc/self"などのシンボリックリンクがOSによって提供されていない環境でも、getpid(2)を用いて自分自身のプロセスIDを調べることによってフルパスを得ることができます。以下に、getpid(2)を用いてフルパスを取得するコードを示します。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    char buf[256];
    char fullpath[1024];
    sprintf(buf, "/proc/%d/exe", getpid());
    ret = readlink(buf, fullpath, 1024)
    assert(ret != -1);
    printf("%s\n", fullpath);
    return 0;
}
```

Win32 APIを利用する

Windows環境にはprocfsはありませんが、Win32 APIを用いることによって自分自身の名

前を簡単に調べることができます。使用するAPIは、実行ファイルのハンドルを取得するGetModuleHandleと、ハンドルからファイル名を取得するGetModuleFileNameです。以下にWin32 APIを利用してフルパスを得るプログラムを示します。

```
#include <windows.h>
#include <stdio.h>
#include <assert.h>

int main() {
    int ret;
    HMODULE h;
    char fullpath[1024];

    h = GetModuleHandle(NULL);
    assert(h);
    ret = GetModuleFileName(h, fullpath, 1024);
    assert(ret != 0);

    printf("%s\n", fullpath);

    return 0;
}
```

getexecname(3C)を利用する

Solarisではまさに実行プロセスのパスを得るためのgetexecname(3C)という関数があります。このように、OSがパス取得を助けてくれる環境では非常に楽にパスを調べることができます。

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    const char *fullpath = getexecname();
    printf("%s\n", fullpath);

    return 0;
}
```

動的ロード情報を利用する

動的ロード情報の中には、実行プロセス自身の情報が含まれている環境が多いため、その中から取得することもできます。動的ロード情報を取得する方法は次のHackで紹介します。この方法は、動的ロード情報がいずれにせよ必要な場合や、この方法以外で手軽にパスが取得できない環境において有効です。

まとめ

実行中のプロセスのパスを調べるさまざまな方法を紹介しました。この方法には決定打がなく、プログラムへの要求によって方法を選択する必要があります。それぞれの方法の要点は以下のようなものです。

- argv[0]と $PATH から調べる方法は、実装と実行時にコストがかかりますが、多くの場合に確実な方法です。
- procfs を用いる方法は、手軽な方法ですが OS 依存性が高くなります。
- Win32 APIやgetexecname(3C)を用いる方法など、OSが協力してくれる環境ではそのAPI を用いるのが簡単です。
- 次のHackで紹介する、動的ロード情報を得る方法によってもパスを得られることが多くあります。

—— Shinichiro Hamaji

HACK #65 ロードしている共有ライブラリをチェックする

本Hackでは、実行中のプロセスがロードしている共有ライブラリを調べる方法を紹介します。このHack はターゲットごとに適切な選択を行う必要があります。

「**[Hack #64]　実行中のプロセスのパス名をチェックする**」では実行ファイルのパスを調べる方法を紹介しましたが、実行時に情報を持っているのは実行ファイルだけではありません。多くの場合、共有ライブラリのパスも同様に知る必要があります。本 Hack では、実行時にロードされている共有ライブラリのパスとロードアドレスを知る方法を紹介します。静的に共有ライブラリを調べる方法は「**[Hack #7]　lddで共有ライブラリの依存関係をチェックする**」を参照して下さい。

ロードアドレス

「**[Hack #6]　静的ライブラリと共有ライブラリ**」で述べられていたように、共有ライブラリは実行の直前にローダによってメモリマップを用いてロードされ、この時までシンボルのアドレスは確定していません。ロードされた共有ライブラリのシンボルのアドレスを特定するためには、ロードされた位置を用いて調整する必要があります。この、ロードされた位置をロードアドレスと呼びます。

Linuxなどでは、procfsを用いてロードされている共有ライブラリとロードアドレスを手軽に調べることができます。例えば以下のようなコマンドを実行すると、

```
% cat /proc/self/maps
```

以下のような出力が得られます。

```
08048000-0804d000 r-xp 00000000 03:06 193783     /bin/cat
0804d000-0804e000 rw-p 00004000 03:06 193783     /bin/cat
0804e000-0806f000 rw-p 0804e000 00:00 0          [heap]
41000000-4101a000 r-xp 00000000 03:06 345981     /lib/ld-2.3.5.so
4101a000-4101b000 r--p 00019000 03:06 345981     /lib/ld-2.3.5.so
4101b000-4101c000 rw-p 0001a000 03:06 345981     /lib/ld-2.3.5.so
4101e000-41141000 r-xp 00000000 03:06 345983     /lib/libc-2.3.5.so
41141000-41143000 r--p 00123000 03:06 345983     /lib/libc-2.3.5.so
41143000-41145000 rw-p 00125000 03:06 345983     /lib/libc-2.3.5.so
41145000-41147000 rw-p 41145000 00:00 0
b7cb8000-b7d92000 r--p 0205b000 03:06 261739     /usr/lib/locale/locale-archive
b7d92000-b7f92000 r--p 00000000 03:06 261739     /usr/lib/locale/locale-archive
b7f92000-b7f93000 rw-p b7f92000 00:00 0
b7fb0000-b7fb1000 rw-p b7fb0000 00:00 0
bfd9a000-bfdb1000 rw-p bfd9a000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

procfsでは、/proc/<process_id>/*を調べることによってプロセスの情報を調べることができますが、「**[Hack #64] 実行中のプロセスのパス名をチェックする**」でも解説したように、Linux環境では/proc/selfが実行中プロセスのプロセスIDへのシンボリックリンクになっています。つまりこの場合は、出力するために実行したcatコマンド自身の情報を出力していることになります。

この表は、左から順に、メモリマップの開始位置―終了位置、メモリ保護属性、ファイル内のオフセット、ファイルのあるデバイスのメジャー番号:マイナー番号、ファイルのiノード、ファイル名となっています。このHackのゴールは、メモリマップの開始位置と、ファイル名を動的に取得することです。

OSがprocfsによってこの情報を提供している場合、このファイルをパースすることもHackを実現する方法の1つとなります。Linuxの例のようにテキスト形式であれば適当にパースしてやっても良いですし、sys/procfs.hのようなヘッダファイルが助けになる場合もあるでしょう。ただし、後述するようにprelinkによってロードアドレスが不要になるケースがあることには注意して下さい。

Linuxでdl_iterate_phdr(3)を利用する

Linuxではdl_iterate_phdr(3)という、まさにこのHackの目的を達成するための関数がglibc内に用意されています。この関数はLinux固有ですが、Linux環境では手軽な解決方法となります。「**[Hack #64] 実行中のプロセスのパス名をチェックする**」同様、ターゲットのOSがHackを手助けする方法を準備してくれている場合は、その方法を使用するのが一番簡単です。

dl_iterate_phdrはコールバックによってライブラリをチェックしていく設計になっています。以下にサンプルコードを示します。

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

static int print_callback(struct dl_phdr_info *info, size_t size, void *data) {
    printf("%08x %s\n", info->dlpi_addr, info->dlpi_name);
    return 0;
}

int main() {
    dl_iterate_phdr(print_callback, NULL);
    return 0;
}
```

このサンプルは、ロードアドレスとライブラリファイル名をただ標準出力に出力するだけのものです。コールバック関数のdata引数はdl_iterate_phdrの第2引数がそのまま渡されます。コールバックの処理にコンテキストの影響を与えたい場合に利用して下さい。また、コールバック関数で0以外を返すと、途中でコールバックを終了します。

以下にFedora Core 4でのサンプルコードの実行結果を示します。

```
% ./a.out
00000000
00000000
00000000 /lib/libc.so.6
00000000 /lib/ld-linux.so.2
```

確かに動的ロードされているライブラリの名前が表示されていますが、不明な点が2点あります。

第一に、最初の2行のファイル名が表示されていない部分は何でしょうか。1行目はこのプロセス自身の情報です。どうやらdl_iterate_phdrでは「**[Hack #64] 実行中のプロセスのパス名をチェックする**」ことはできないようです。2行目は、linux-gate.so.1という仮想共有ライブラリで、カーネルによって提供されているものなので、ファイル名が表示できていません。先ほどの/proc/self/mapsの内容の中の[vdso]という空間がこれに対応しています。このファイルについての解説は **[Hack #59]** を参照して下さい。

第二に、なぜロードされている共有ライブラリのロードアドレスが2つとも0なのでしょうか。先ほどの/proc/self/mapsを参照するに、0x41000000付近の値でないとおかしいはずです。これは、この環境ではprelinkが実行されていて、すでにシンボルの値にロードアドレス分のオフセットが加わっているからです。prelinkについての詳細な情報は「**[Hack #85] prelinkでプログラムの起動を高速化する**」を参照して下さい。

以下に、prelink使用環境でないDebian GNU/Linuxでの実行結果の例を示します。ロードアドレスが表示されていることがわかります。

```
% ./a.out
00000000
00000000
4001d000 /lib/tls/libc.so.6
40000000 /lib/ld-linux.so.2
```

dlinfo(3) を利用する

FreeBSD や Solaris では、dlinfo 関数と RTLD_SELF、RTLD_DI_LINKMAP フラグを組み合わせて使用することによってこの Hack を実現することができます。dlinfo(3)は dlopen で開いたハンドルと、取得した情報を指定して情報を得る関数ですが、RTLD_SELFを使用することによって自分自身のハンドルを指定することもできます。また、RTLD_DI_LINKMAPを指定することによって共有ライブラリのリストを返すように要求します。以下にサンプルコードを示します。

```
#include <stdio.h>
#include <dlfcn.h>
#include <link.h>
#include <assert.h>

int main() {
    struct link_map *lmap;
    int ret = dlinfo(RTLD_SELF, RTLD_DI_LINKMAP, &lmap);
    assert(ret == 0);
    while (lmap) {
        printf("%08x %s\n", lmap->l_addr, lmap->l_name);
        lmap = lmap->l_next;
    }
    return 0;
}
```

以下は FreeBSD での実行例です。

```
% ./a.out
08048000 ./a.out
28067000 /usr/lib/libc.so.4
28049000 /usr/libexec/ld-elf.so.1
```

以下は Solaris での実行例です。

```
% ./a.out
00010000 a.out
ff3fa000 /usr/lib/libdl.so.1
ff280000 /usr/lib/libc.so.1
ff3a0000 /usr/platform/SUNW,Sun-Fire-V210/lib/libc_psr.so.1
```

dlfcn.hには、他にもこのHackの実現を助けることができそうな関数があります。NetBSDやOpenBSDのヘッダ内に記述されているdlctl(3)はこのHackを実現することができそうに見えます。しかし、残念ながら筆者が確認したバージョンではまだ未実装である旨、ヘッダに記述されていました。

また、共有ライブラリ名と関数の名前が何か1つでもわかっていれば、dlopen、dlsym、dladdrの組み合わせによってロードアドレスを調べることができます。

Win32 APIを利用する

WindowsもWin32 APIを通じて、このHackを実現する方法を提供しています。tlhelp32.h内のCreateToolhelp32Snapshotでプロセスのスナップショットを取得し、Module32FirstとModule32Nextでモジュールハンドルを巡回します。以下にサンプルコードを示します。

```
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <tlhelp32.h>

void print_module(MODULEENTRY32 *me) {
    char buf[1024];
    int ret = GetModuleFileName(me->hModule, buf, 1024);
    assert(ret);
    printf("%08x %s\n", (int)me->modBaseAddr, buf);
}

int main() {
    HANDLE ss;
    MODULEENTRY32 me;
    int ret;

    ss = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, 0);
    assert(ss);
    me.dwSize = 1024;
    ret = Module32First(ss, &me);
    assert(ret);

    print_module(&me);
    for (;;) {
        me.dwSize = 1024;
        if (!Module32Next(ss, &me)) {
            break;
        }
        print_module(&me);
    }

    return 0;
}
```

以下に実行結果の例を示します。

```
% ./a.exe
00400000 D:\wrk\binhack\a.exe
7f7b0000 C:\windows\system\msvcrt.dll
7fd20000 c:\windows\system\kernel32.dll
7ffa0000 c:\windows\system\ntdll.dll
```

1行目では自分自身のフルパスが得られているため、「**[Hack #64] 実行中のプロセスのパス名をチェックする**」方法の代用としても利用できることがわかります。

Mac OS X で dyld(3)を利用する

Mac OS X では、dyld(3)を利用するのが簡単です。正確には dyld(3)という関数は存在しないので、_dyld プレフィクスで始まる関数群を利用するのですが、このマニュアルは man 3 dyld で見ることができるため、このように表記しています。

以下にサンプルコードを示します。_dyld_image_count で共有ライブラリの数を取得して、その回数だけ _dyld_get_image_name と _dyld_get_image_vmaddr_slide を呼んでいます。

```
#include <mach-o/dyld.h>

int main() {
    int num, i;
    num = _dyld_image_count();
    for (i = 0; i < num; i++) {
        printf("%08x %s\n",
                _dyld_get_image_vmaddr_slide(i), _dyld_get_image_name(i));
    }
    return 0;
}
```

以下に実行結果を示します。

```
% ./a.out
00000000 ./a.out
00000000 /usr/lib/libSystem.B.dylib
00000000 /usr/lib/system/libmathCommon.A.dylib
```

Mac OS X は、prebinding という機構によって、prelink と同等のことを行っています。そのため、またしてもロードアドレスがすべて 0 となっています。

また、1 行目に表示されている ./a.out は、argv[0]と同じ値となっています。そのため、「**[Hack #64] 実行中のプロセスのパス名をチェックする**」方法としては機能しません。

まとめ

実行中のプロセスのロードしている共有ライブラリを調べる方法を紹介しました。このHackは環境ごとに実現方法がまったく異なり、実に混沌としています。ターゲットごとに適切に選択する必要があります。

—— Shinichiro Hamaji

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

pmapコマンドや/proc/<pid>/mapsファイルを使って、各プログラムのメモリ(ヒープ、スタックなど)がどのように使われているか確認することができます。

プロセスが起動すると、その実行バイナリファイルや、動的ローダによってロードされた共有ライブラリファイルが、そのプロセスの仮想メモリ空間の一部分としてマップされます。どんなファイルがマップされているかは、「**[Hack #65] ロードしている共有ライブラリをチェックする**」で解説しています。そこで、本Hackでは、その中でもプロセスが使用している仮想メモリの範囲がどうなっているのかについて調べてみましょう。

pmapコマンドを使う

LinuxのprocpsパッケージやSolarisの標準コマンドにはpmapコマンドが入っています。このコマンドを使用することで、プロセスが使用中の仮想メモリマップの状態を表示できます。使い方は、pmapコマンドの引数にプロセスIDを指定します。なお、Linuxの場合、pmapコマンドは実は/proc/<pid>/mapsファイルからマッピング情報を取得し、ユーザにわかりやすい形へ変換して表示しているだけです。そのため、/proc/<pid>/mapsファイルを直接表示させても同様な情報を得ることができます。

例として、指定時間だけ停止して終了する/bin/sleepコマンドを動かしている間に、pmapコマンドをi386アーキテクチャのLinux 2.6.15カーネル(Debian/GNU/Linux sarge)上で実行した様子を示します。

```
% /bin/sleep 10000 &
[1] 24039
% ps auxw | grep /bin/sleep
gotom    24039  0.0  0.0   3892   596 pts/13   S    10:48   0:00 /bin/sleep 10000
% pmap 24039
24039:   /bin/sleep 10000
08048000     16K r-x--  /bin/sleep
0804c000      4K rw---  /bin/sleep
0804d000    132K rw---    [ anon ]
b7b69000   2048K r----  /usr/lib/locale/locale-archive
```

```
b7d69000      8K rw---   [ anon ]
b7d6b000     60K r-x--  /lib/tls/i686/cmov/libpthread-2.3.5.so
b7d7a000      8K rw---  /lib/tls/i686/cmov/libpthread-2.3.5.so
b7d7c000      8K rw---   [ anon ]
b7d7e000   1220K r-x--  /lib/tls/i686/cmov/libc-2.3.5.so
b7eaf000      4K r----  /lib/tls/i686/cmov/libc-2.3.5.so
b7eb0000     12K rw---  /lib/tls/i686/cmov/libc-2.3.5.so
b7eb3000      8K rw---   [ anon ]
b7eb5000     28K r-x--  /lib/tls/i686/cmov/librt-2.3.5.so
b7ebc000      8K rw---  /lib/tls/i686/cmov/librt-2.3.5.so
b7ebe000    140K r-x--  /lib/tls/i686/cmov/libm-2.3.5.so
b7ee1000      8K rw---  /lib/tls/i686/cmov/libm-2.3.5.so
b7f00000      4K rw---   [ anon ]
b7f01000     84K r-x--  /lib/ld-2.3.5.so
b7f16000      8K rw---  /lib/ld-2.3.5.so
bff01000     84K rw---   [ stack ]
ffffe000      4K -----   [ anon ]
 total     3896K
```

pmapコマンドの詳細情報を表示させるオプションを使うことで、さらに細かい情報を知ることが可能です。

pmapコマンドが出力した1列目と2列目から、そのプロセスの仮想メモリ中でどの範囲が使用されているかを知ることができます。もし、この範囲外へメモリアクセスを行うと、セグメンテーションフォールトなどのシグナルが発生します。また、3列目ではその範囲へのアクセスパーミッションがわかります。ファイルのパーミッションと同様に、rとなっていれば読み込み、wとなっていれば書き込み、xとなっていれば実行が、それぞれ可能であることを示します。そして、4列目ではその仮想メモリの使用目的を知ることができます。それでは、これらの行の中身をもう少し細かく見ていきましょう。

まず、実行ファイルの/bin/sleepについてです。これは出力の2行目と3行目にあります。r-xとなっているものが16KB分、rw-となっているものが4KB分それぞれ存在しています。これは、readelf -Sで表示されるアドレスと一致した範囲にあります。

```
% readelf -S /bin/sleep
There are 24 section headers, starting at offset 0x3498:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 000148 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048290 000290 0002b0 10   A  5   1  4
  [ 5] .dynstr           STRTAB          08048540 000540 0001f5 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048736 000736 000056 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         0804878c 00078c 000080 00   A  5   2  4
  [ 8] .rel.dyn          REL             0804880c 00080c 000028 08   A  4   0  4
```

```
[ 9] .rel.plt          REL             08048834 000834 000118 08   A  4  11  4
[10] .init             PROGBITS        0804894c 00094c 000017 00  AX  0   0  4
[11] .plt              PROGBITS        08048964 000964 000240 04  AX  0   0  4
[12] .text             PROGBITS        08048bb0 000bb0 001cf0 00  AX  0   0 16
[13] .fini             PROGBITS        0804a8a0 0028a0 00001b 00  AX  0   0  4
[14] .rodata           PROGBITS        0804a8c0 0028c0 000958 00   A  0   0 32
[15] .data             PROGBITS        0804c218 003218 000024 00  WA  0   0  4
[16] .eh_frame         PROGBITS        0804c23c 00323c 000004 00   A  0   0  4
[17] .dynamic          DYNAMIC         0804c240 003240 0000d8 08  WA  5   0  4
[18] .ctors            PROGBITS        0804c318 003318 000008 00  WA  0   0  4
[19] .dtors            PROGBITS        0804c320 003320 000008 00  WA  0   0  4
[20] .jcr              PROGBITS        0804c328 003328 000004 00  WA  0   0  4
[21] .got              PROGBITS        0804c32c 00332c 0000a0 04  WA  0   0  4
[22] .bss              NOBITS          0804c3e0 0033e0 00014c 00  WA  0   0 32
[23] .shstrtab         STRTAB          00000000 0033e0 0000b5 00      0   0  1
```

以上の表示から、上記 /bin/sleep プログラムのうち、0x08048000-0x0804bfff のアドレス範囲には、セクション.interp～.rodataが割り当ててあることがわかります。これらの領域は、書き込みはできない代わりに実行は許可されています。また、0x0804c000-0x0804cfff のアドレス範囲には、書き込みはできるが実行はできないセクション.data～.bssが存在しています。.plt、.got、.text、.rodata、.data といったセクションがどちらかの領域に存在しています。

なお、これは x86 の例ですので、ページサイズ 4KB ごとに領域が割り当てられています。アーキテクチャが変わることによって、これらのアドレス範囲も変わってきます。

その他のファイル名として、動的ローダ ld-2.3.5.so や共有ライブラリ libc-2.3.5.so を見ることができます。もちろんこれらのファイルは共有ライブラリですから、readelf -S で見ても、上記のように固定のアドレスが設定されているわけではありません。しかし、それぞれいくつか領域ごとにアドレス範囲が別々に設定されているのは/bin/sleepの場合と同様です。

プログラムが確保したメモリを pmap コマンドで確認する

続いて、プログラムが確保している anon や stack といった領域について、どのように使われているか詳しく調べるために、メモリ獲得を行うサンプルプログラムを実行した例を見てみましょう。以下がサンプルプログラムです。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define SLEEP 7

void do_malloc(size_t size)
{
  void *mem;
  mem = malloc(size * 1024);
```

```
  if (mem == NULL) {
    printf("memory exhausted\n");
    exit(1);
  }
  printf("%u KB allocated\n", size);
  sleep(SLEEP);
}

#define STACKSIZE (1024 * 1024)
void do_stack(void)
{
  int stack[STACKSIZE];
  stack[STACKSIZE - 1] = 0;
  printf("stack 1MB allocated\n");
  sleep(SLEEP);
}

int main(void)
{
  printf("process %u started\n", getpid());
  sleep(SLEEP);
  do_malloc(8);      /* 8KB 確保 */
  do_malloc(100);    /* 100KB 確保 */
  do_malloc(100);    /* 100KB 確保 */
  do_malloc(1024);   /* 1MB 確保 */
  do_stack();        /* スタックを大きくする */
  exit(0);
}
```

このプログラムは、起動後7秒ごとに停止しながら、さまざまなサイズのmalloc()を実行します。そして実行中、横でpmapコマンドを実行してどの領域が変化するのを観察します。

```
% ./malloc
process 30939 started
8 KB allocated
100 KB allocated
100 KB allocated
1024 KB allocated
stack 1MB allocated
```

このとき、straceを実行すると次のように表示されます(長いので適宜省略しています)。

```
% strace ./malloc
execve("./malloc", ["./malloc"], [/* 31 vars */]) = 0
uname({sys="Linux", node="celesta", ...}) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fbe000
```

```
open("/etc/ld.so.cache", O_RDONLY)      = 3
write(1, "process 30956 started\n", 22process 30956 started  ) = 22
nanosleep({7, 0}, {7, 0})               = 0
brk(0)                                  = 0x804a000
brk(0x806d000)                          = 0x806d000
write(1, "8 KB allocated\n", 15)        = 15
nanosleep({7, 0}, {7, 0})               = 0
write(1, "100 KB allocated\n", 17)      = 17
nanosleep({7, 0}, {7, 0})               = 0
brk(0x809f000)                          = 0x809f000
write(1, "100 KB allocated\n", 17)      = 17
nanosleep({7, 0}, {7, 0})               = 0
mmap2(NULL, 1052672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7d68000
write(1, "1024 KB allocated\n", 18)     = 18
nanosleep({7, 0}, {7, 0})               = 0
write(1, "stack 1MB allocated\n", 20)   = 20
nanosleep({7, 0}, {7, 0})               = 0
munmap(0xb7fbd000, 4096)                = 0
```

動作中にどのようにpmapコマンドの出力が変化するかを見ていきましょう。紙面にかぎりがあるので変化があったエントリだけ注目します。まず、8KB獲得すると、以下のエントリが新たに出現します。これは、malloc()の延長線上で呼ばれるbrk()呼び出しによって現れるヒープ領域です。

```
0804a000    140K rw---    [ anon ]
b7dd3000      4K rw---    [ anon ]
```

続いて100KBを2回獲得すると以下のように変化します。これはヒープが足りなくなったためにやはりbrk()システムコールによって拡張された領域です。

```
0804a000    340K rw---    [ anon ]
b7dd3000      4K rw---    [ anon ]
```

さらに1MBを獲得すると、次のようになります。これは、ヒープが拡大せず、変わりに0xb7cd2000から始まるアドレスが増えています。これは、glibcのmallocアロケータが巨大なメモリ領域はanonymous mmapによって取得したためです。

```
0804a000    340K rw---    [ anon ]
b7cd2000   1032K rw---    [ anon ]
```

今度はスタックを獲得すると、次のように変化します。

```
bf73c000   4104K rw---    [ stack ]
```

スタック領域が増えました。

まとめ

pmapコマンドや/proc/<pid>/mapsファイルを使って、各プログラムのメモリ(ヒープ、スタックなど)がどのように使われているか確認できることを紹介しました。このHackは共有ライブラリにも同じようにあてはめて考えることができます。

—— Masanori Goto

HACK #67 libbfdでシンボルの一覧を取得する

本Hackでは、libbfdの紹介と、シンボルの名前とアドレスの一覧を取得する方法、そしてアドレスからファイル名と行情報を取得する方法を紹介します。

GNU binutilsは、そのほとんどがlibbfdというライブラリのインターフェースコマンドとして実現されています。BFDとはBinary File Descriptorの略で、その名の通りバイナリファイルを読み書きするためのライブラリです。

本Hackでは、libbfdの簡単な紹介と、使用法の一例として、ネイティブバイナリでのリフレクションを実現するために必須となる、シンボルの名前とアドレスの一覧を取得する方法を紹介します。

libbfdの概要

libbfdは前述した通り、GNU binutilsの一部として開発されているライブラリです。GPLのもとで配布されています。

libbfdは、Fedora Core 4ではbinutilsパッケージの一部として、Debian GNU/Linuxではbinutils-devパッケージに含まれています。パッケージとして入っていない環境であっても、binutilsを自分でコンパイルしてインストールすれば利用することができます。

libbfdは、GCCと同時に使うことが想定されているため当然ですが、非常に多数のバイナリフォーマット、アーキテクチャがサポートされています。多くの場合、これらのフォーマットやアーキテクチャによる差異は、libbfd内で吸収されて、ユーザはさまざまな環境で動くソフトウェアを1つのコードで書くことができます。

libbfdのドキュメントとしては、http://www.sra.co.jp/wingnut/bfd/bfd-ja.htmlにリファレンスの翻訳があります。また、使用法で不明な点がある場合は、binutils内の同一処理部分を参考にすると良いサンプルとなるでしょう。例えばこのHackの内容はbinutils内のnm(1)を参考にしています。

minisymbol 系の API を利用してシンボル一覧を得る

libbfd にはシンボルテーブルを得る API は 2 つ用意されています。まずは、一度にシンボルテーブルを作らないため省メモリな minisymbol 系の API を紹介します。以下にサンプルコードを示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <bfd.h>

void dump_symbols(const char *filename) {
    bfd *abfd;
    asymbol *store;
    char *p;
    void *minisyms;
    int symnum, i;
    size_t size;
    /* 動的シンボルを取得する場合は 1 に */
    int dyn = 0;
    int ret;

    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
        bfd_close(abfd);
        return;
    }

    store = bfd_make_empty_symbol(abfd);

    symnum = bfd_read_minisymbols(abfd, dyn, &minisyms, &size);
    assert(symnum >= 0);

    p = (char *)minisyms;
    for (i = 0; i < symnum; i++) {
        asymbol *sym = bfd_minisymbol_to_symbol(abfd, dyn, p, store);
        const char *name = bfd_asymbol_name(sym);
        int value = bfd_asymbol_value(sym);
        printf("%08x %s\n", value, name);
        p += size;
    }

    free(minisyms);
    bfd_close(abfd);
}
```

```
int main(int argc, char *argv[]) {
    /* 「[Hack #64] 実行中のプロセスのパス名をチェックする」と組み合わせるとベター */
    dump_symbols(argv[0]);

    return 0;
}
```

プログラム内コメントで解説したように、nm -D相当の、動的シンボルが欲しい場合はdynに1をセットして下さい。これはstripされた共有ライブラリ中のシンボルを探す場合などに有効です。

symtab系のAPIを利用する

bfd_get_symtab_upper_boundとbfd_canonicalize_symtab関数を使用する方法もあります。このAPIでは一気にasymbol構造体の配列を取得します。同じプログラムではあまり面白くないですから、今回はシンボル一覧に加えてファイル名と行数も取得してみます。以下に自身のシンボル一覧と、その定義されているファイル名と行数を表示するプログラムを示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <bfd.h>

void dump_symbols(const char *filename) {
    bfd *abfd;
    long storage;
    asymbol **syms;
    int symnum;
    int i;
    int ret;

    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
        bfd_close(abfd);
        return;
    }

    storage = bfd_get_symtab_upper_bound(abfd);
    assert(storage >= 0);
    if (storage) syms = (asymbol **)malloc(storage);

    symnum = bfd_canonicalize_symtab(abfd, syms);
```

```
    assert(symnum >= 0);

    for (i = 0; i < symnum; i++) {
        asymbol *sym = syms[i];
        int value = bfd_asymbol_value(sym);
        const char *file, *name;
        int lineno;
        asection *dbgsec = bfd_get_section_by_name(abfd, ".debug_info");

        ret = bfd_find_nearest_line(abfd, dbgsec, syms, value,
                                    &file, &name, &lineno);

        if (ret && file && name) {
            printf("%08x %s (%s:%d)\n", value, name, file, lineno);
        }
        else {
            name = bfd_asymbol_name(sym);
            printf("%08x %s\n", value, name);
        }
    }

    free(syms);
    bfd_close(abfd);
}

int main(int argc, char *argv[]) {
    /* 「[Hack #64] 実行中のプロセスのパス名をチェックする」と組み合わせるとベター */
    dump_symbols(argv[0]);

    return 0;
}
```

bfd_find_nearest_lineを用いてファイル名と行情報を取得しています。この情報は.debug_infoセクションにあるため、bfd_get_section_by_nameでセクション構造体を取得しています。これは「**[Hack #15] addr2line でアドレスからファイル名と行番号を取得する**」で紹介したaddr2line(1)相当の処理です。

動的なシンボルを取得したい場合は、bfd_get_symtab_upper_boundとbfd_canonicalize_symtabをbfd_get_dynamic_symtab_upper_boundとbfd_canonicalize_dynamic_symtabに変更すればうまくいきます。

まとめ

本 Hack では、libbfd の紹介と、シンボルの名前とアドレスの一覧を取得する方法と、アドレスからファイル名と行情報を取得する方法を紹介しました。libbfdはバイナリファイルの読み書きに便利なライブラリです。GNU binutils相当の処理を実行時に行いたい場合は有用となるでしょう。

—— Shinichiro Hamaji

HACK #68 C++ のシンボルを実行時にデマングルする

本HackではC++のシンボルを実行時にデマングルする方法としてcplus_demangle()、abi::__cxa_demangle()を紹介します。

C++コンパイラはシンボルが一意の名前を持つように名前マングル（name mangling）と呼ばれる処理を行います。本Hackでは実行時にC++のシンボルをデマングル（demangle）する方法を紹介します。コマンドラインからデマングルする方法については「**[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。

マングルの方法はコンパイラ依存です。同じコンパイラでもバージョンによってマングルの方法が異なることがあります。たとえばGCC 3.xではint foo(int)を_Z3fooiに、int foo(const char*)を_Z3fooPKcのようにマングルしますが、GCC 2.95ではそれぞれfoo__Fi、foo__FPCcとなります。

実行時にデマングル

C++のプログラムの中からデマングルするには2つの方法があります。1つはbinutilsのlibibertyに含まれるcplus_demangle()を使う方法、もう1つはlibstdc++に含まれるabi::__cxa_demangle()を使う方法です。後者が使える環境であれば後者を使う方法がよいでしょう。

```
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>
using namespace std;

// From binutils/include/demangle.h
#define DMGL_PARAMS      (1 << 0)  /* Include function args */
#define DMGL_ANSI        (1 << 1)  /* Include const, volatile, etc */
#define DMGL_VERBOSE     (1 << 3)  /* Include implementation details.  */
#define DMGL_TYPES       (1 << 4)  /* Also try to demangle type encodings. */
extern "C" char *cplus_demangle(const char *mangled, int options);

int main() {
    // using libiberty
    int options = DMGL_PARAMS | DMGL_ANSI | DMGL_TYPES;
    cout << cplus_demangle("_Z3fooPKc",  options) << endl;
    cout << cplus_demangle("i",  options) << endl;

    // using libstdc++
    int status;
    cout << abi::__cxa_demangle("_Z3fooPKc", 0, 0, &status) << endl;
    cout << abi::__cxa_demangle("i", 0, 0, &status) << endl;
    return 0;
}
```

実行結果は次のようになります。

```
% g++ test.cpp -liberty && ./a.out
foo(char const*)
int
foo(char const*)
int
```

cplus_demangle()はlibiberty.aに含まれていますが、libiberty.hに宣言されていないので、binutils/include/demangle.h から必要な宣言をコピーして使う必要があります。また、libiberty.a をリンクする必要もあります。なお、cplus_demangle()は malloc したメモリを返すため、上のコードにはメモリリークがあります。

abi::__cxa_demangle()はcxxabi.hをインクルードすれば使えますが、GCC 2.95など、古いGCC には含まれていないのが難点です。abi::__cxa_demangle()のコードは binutils から GCC に取り入れられているため、処理の中身はほぼ同じと考えていいと思います。なお、abi::__cxa_demangle(mangled, 0, 0, &status)のように呼び出した場合、結果は malloc したメモリで返されるため、上のコードにはメモリリークがあります。

typeid と連携させる

C++ではtypeid演算子を用いて実行時に型の情報を得ることができます。さらに、typeid()が返す type_info オブジェクト(のリファレンス)に対して name()メンバ関数を呼ぶと、型の名前を文字列で得られます。

GCC(少なくとも3.3)では、この型の名前がマングルされているので、読みやすい形式にするにはデマングルする必要があります。

```
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>
using namespace std;
struct Foo {
    virtual ~Foo() {};
};
struct Bar : public Foo {
    virtual ~Bar() {};
};

// メモリリークしています
char* demangle(const char *demangle) {
    int status;
    return abi::__cxa_demangle(demangle, 0, 0, &status);
}

int main() {
```

```
    Bar bar;
    Foo *p = &bar;
    cout << typeid(int).name() << endl;
    cout << typeid(Foo).name() << endl;
    cout << typeid(p).name() << endl;
    cout << typeid(*p).name() << endl;
    cout << endl;

    cout << demangle(typeid(int).name()) << endl;
    cout << demangle(typeid(Foo).name()) << endl;
    cout << demangle(typeid(p).name()) << endl;
    cout << demangle(typeid(*p).name()) << endl;
    return 0;
}
```

実行結果は次のようになります。

```
% g++ test.cpp && ./a.out
i
3Foo
P3Foo
3Bar

int
Foo
Foo*
Bar
```

FooとBarに仮想デストラクタをつけているのは、仮想関数が1つもないとtypeid(*p).name()が"Bar"ではなく、なぜか"Foo"を返したためです。

まとめ

本HackではC++のシンボルを実行時にデマングルする方法としてcplus_demangle()、abi::__cxa_demangle()を紹介しました。これらを押さえておけば、マングルされたシンボルもこわくありません。

—— Satoru Takabayashi

HACK #69 ffcallでシグネチャを動的に決めて関数を呼ぶ

ffcallやlibffiを用いると実行時にシグネチャを指定して関数が実行できます。

関数アドレスを使って関数を呼び出すことを考えます。コンパイル時にシグネチャが決まっていればただ単にキャストして実行してやればよいでしょう。しかし、例えば動的情報

を豊富に持つネイティブバイナリを実行するプログラム言語を作成した場合など、関数シグネチャを実行時に決定したい場合もあります。本 Hack では ffcall というライブラリを用いてこれを実現する方法を紹介します。

ffcall

ffcall は、Foreign Function CALL の略です。ffcall は GPL2 で配布され、執筆時のバージョンは 1.10 です。

ffcallはGNUstep（Objective-C用のシステムの基本的なクラスを集めたライブラリ）に用いられているライブラリです。Objective-Cは、まさに動的情報を豊富に持ちながらネイティブバイナリで実行するプログラム言語ですので、リフレクションのような機構を実現するためにこういったライブラリが用いられていることは不思議ではありません。

ここでは述べませんが、ffcall にはトランポリンを実現する API も入っています。トランポリンについては、「**[Hack #32] GCC が生成したコードによる実行時コード生成**」を参照して下さい。

早速簡単なサンプルを見てみましょう。

```
#include <avcall.h>
#include <stdio.h>

/* 文字列中 n 番目の文字を返す関数 */
char nth(const char *str, int i) {
    return str[i-1];
}

int main() {
    int n = 5;
    const char *msg = "binary";
    char ret;
    av_alist alist;

    /* 通常の呼び出し */
    printf("NORMAL: %c\n", nth(msg, n));

    /* ffcall を用いた呼び出し */
    av_start_char(alist, &nth, &ret);
    av_ptr(alist, const char *, msg);
    av_int(alist, n);
    av_call(alist);

    printf("FFCALL: %c\n", ret);

    return 0;
}
```

nthは文字列中のn番目の文字を返す関数です。これをまず通常の関数呼び出しで呼び出した後、ffcallを用いて関数呼び出ししています。最初のav_start_charでalistにchar返り値の関数nthとその返り値の保存先retをセットしています。次のav_ptr、av_intで引数を型を指定しながら設定し、av_callで関数を呼び出します。そして、結果がretに入っているはずですのでそれを出力します。実行結果を以下に示します。

```
% ./a.out
NORMAL: r
FFCALL: r
```

見事、通常の呼び出しと同様の結果を得られていることがわかります。

実行時にシグネチャ変更

これだけではあまり面白くありませんので、実際に実行時にシグネチャを指定してみましょう。以下のサンプルは、コマンドライン引数に応じて関数シグネチャを指定して実行します。第1引数が関数名で、以降はsかiで文字列か整数かだけを指定してから、その次の引数に内容を入れていくことにします。他の型はここではサポートしません。

```
#include <avcall.h>
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char *argv[]) {
    int ret;
    av_alist alist;
    void *dlh;
    void *fp;
    int i;

    if (argc < 2) return 1;

    dlh = dlopen(argv[0], RTLD_LAZY);
    fp = dlsym(dlh, argv[1]);

    av_start_int(alist, fp, &ret);

    for (i = 2; i < argc; i += 2) {
        /* string */
        if (argv[i][0] == 's') {
            av_ptr(alist, char *, argv[i+1]);
        }
        /* int */
        else if (argv[i][0] == 'i') {
            av_int(alist, atoi(argv[i+1]));
        }
```

```
    }

    av_call(alist);

    printf("\nRESULT: %d\n", ret);

    return 0;
}
```

この例では、argv[0]を自身の名前であると仮定してそれを開き、dlsymでargv[1]で指定した関数名からアドレスを引いてきています。これらを本格的に行う方法はここまでのHackで述べてきた通りです。さて、いろいろと実験をしてみましょう。

```
% ./a.out puts s "hello world"
hello world

RESULT: 12
```

まずは hello world です。きちんと表示されました。結果の 12 は puts の出力文字数です。次は int も使用してみます。

```
% ./a.out printf s %d i 7
7
RESULT: 1
```

printf("%d", 7)を呼び出してみました。これも正しい結果が出ています。最後に、もう少し引数を増やしてみます。

```
% ./a.out printf s %d+%d=%s i 3 i 4 s SEVEN
3+4=SEVEN
RESULT: 9
```

引数を増やしてもこのサンプルはうまく動いているようです。

libffi

このような機能を実現するライブラリはffcallだけではありません。同じく動的な型システムを持ち、ネイティブバイナリを実行する言語環境である、GCJ（GCCのJavaバインディング）で用いられているlibffiによっても実行時にシグネチャを指定して関数実行ができます。

ffi は Foreign Function Interface の略で、MIT ライセンスで配布されており、GCC の一部として GCJ のソースコードに同梱されています。

libffiはffcallと非常に似ているため、ここではffcallの最初の例をlibffiで実装するだけとします。

```
#include <ffi.h>
#include <stdio.h>

char nth(const char *str, int i) {
    return str[i-1];
}

int main() {
    int n = 5;
    const char *msg = "binary";

    ffi_cif cif;
    int arg_num;
    ffi_type *arg_types[2];
    void *arg_values[2];
    ffi_arg ret;
    ffi_type *ret_type;

    /* 通常の呼び出し */
    printf("NORMAL: %c\n", nth(msg, n));

    /* libffi を用いた呼び出し */
    ret_type = &ffi_type_schar;
    arg_num = 2;
    arg_types[0] = &ffi_type_pointer;
    arg_values[0] = &msg;
    arg_types[1] = &ffi_type_sint;
    arg_values[1] = &n;

    ffi_prep_cif(&cif, FFI_DEFAULT_ABI, arg_num, ret_type, arg_types);
    ffi_call(&cif, FFI_FN(nth), &ret, arg_values);

    printf("LIBFFI: %c\n", ret);

    return 0;
}
```

libffiでは、引数がいくつあっても、ffi_prep_cifで準備して、ffi_callで呼び出す、2つのAPIしか使用しません。その代わりに配列に引数型情報などをセットして準備をしておく必要があります。

他のHackと組み合わせる

「**[Hack #68] C++のシンボルを実行時にデマングルする**」と組み合わせると、例えばC++ではマングリングされたシンボルに引数の情報があるため、これらの情報をうまく用いればシグネチャの指定が楽になるかもしれません。これは、sやiで引数型を指定しなくてすむ可能性があるということを意味します。残念ながら、GCCのC++マングリングルールでは、返り値の型はシンボルには埋め込まれません。これらの情報を確実に取得したい場合は

「**[Hack #70]　libdwarf でデバッグ情報を取得する**」を参照すると良いでしょう。

まとめ

ffcallやlibffiを用いると実行時にシグネチャを指定して関数が実行できます。このHackは主に言語環境に効果を発揮するでしょう。

—— Shinichiro Hamaji

HACK #70 libdwarf でデバッグ情報を取得する

本Hackでは、GCC環境などで使用されるデバッグ情報であるDWARF2を扱うライブラリであるlibdwarfを紹介します。

DWARF2

DWARF2はデバッグ情報をオブジェクトファイルに保存するフォーマットです。「**[Hack #8] readelf で ELF ファイルの情報を表示する**」で解説されているように、DWARF2 情報はreadelf の -w オプションで見ることができます。

DWARF2では、.debugで始まる複数のセクションにそれぞれ異なった種類のデバッグ情報を保存します。ここでは型の情報や関数や変数の型、名前、行情報などを含むセクションである、.debug_info セクションを扱います。

この情報はコンパイル単位(compile unit、libdwarf では cu と省略されています)をルートとしたツリー構造になっていて、階層構造は基本的にプログラムの階層構造に対応したものとなっています。

libdwarf のインストール

libdwarf は http://reality.sgiweb.org/davea/dwarf.html で配布されており、ライセンスはLGPL です。執筆時のバージョンは 20051201 です。

libdwarf のコンパイルには libelf が必要です。Debian では libelfg0-dev パッケージとlibdwarf-dev パッケージを、Fedora Core 4 では elfutils-libelf-devel をインストールの上、自前でコンパイルすることになります。

また、libdwarfにはdwarfdumpというDWARF2情報をダンプするツールも付属しています。readelf -w の代替として、あるいはlibdwarfの使用法を調べるサンプルコードとして使用できます。

変数名一覧を調べるサンプル

以下にlibdwarfを用いて実行ファイルの情報から宣言されている変数とその行数を表示するサンプルを示します。

```
#include <libdwarf/libdwarf.h>
#include <libdwarf/dwarf.h>
#include <libelf.h>

#include <stdio.h>
#include <fcntl.h>
#include <assert.h>

/* 変数名を表示しつつ再帰的に DWARF2 情報を見ていく */
static void process_one_die(Dwarf_Debug dbg, Dwarf_Die die, int d) {
    Dwarf_Error err;
    int ret;

    while (1) {
        Dwarf_Half tag;
        Dwarf_Die child;

        ret = dwarf_tag(die, &tag, &err);
        assert(ret == DW_DLV_OK);

        if (tag == DW_TAG_variable ||
            tag == DW_TAG_formal_parameter)
        {
            Dwarf_Attribute attr;
            Dwarf_Unsigned line;
            char *str;

            ret = dwarf_attr(die, DW_AT_decl_line, &attr, &err);
            /* 特殊な変数では行情報がない場合もある */
            if (ret == DW_DLV_NO_ENTRY) goto next;
            assert(ret == DW_DLV_OK);
            ret = dwarf_formudata(attr, &line, &err);
            assert(ret == DW_DLV_OK);

            ret = dwarf_attr(die, DW_AT_name, &attr, &err);
            assert(ret == DW_DLV_OK);
            ret = dwarf_formstring(attr, &str, &err);
            assert(ret == DW_DLV_OK);

            printf("%d: %s\n", (int)line, str);
        }

    next:
        ret = dwarf_child(die, &child, &err);
        assert(ret != DW_DLV_ERROR);
        if (ret == DW_DLV_OK) {
```

```
            process_one_die(dbg, child, d+1);
        }

        ret = dwarf_siblingof(dbg, die, &die, &err);
        if (ret == DW_DLV_NO_ENTRY) break;
        assert(ret == DW_DLV_OK);
    }
}

static void process_one_file(Elf *elf, const char *filename) {
    /* DWARF2 情報を取得する */

    Dwarf_Debug dbg;
    Dwarf_Die die;
    Dwarf_Error err;
    int ret;

    Dwarf_Unsigned cu_header_length = 0;
    Dwarf_Unsigned abbrev_offset = 0;
    Dwarf_Half version_stamp = 0;
    Dwarf_Half address_size = 0;
    Dwarf_Unsigned next_cu_offset = 0;

    ret = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL, &dbg, &err);
    assert(ret == DW_DLV_OK);

    while ((ret =
            dwarf_next_cu_header(dbg, &cu_header_length, &version_stamp,
                                 &abbrev_offset, &address_size,
                                 &next_cu_offset, &err))
           == DW_DLV_OK)
    {
        ret = dwarf_siblingof(dbg, NULL, &die, &err);

        if (ret == DW_DLV_OK) {
            /* 取得した DWARF2 情報を次の関数へ */
            process_one_die(dbg, die, 0);
        }
        else if (ret == DW_DLV_NO_ENTRY) {
            continue;
        }
        assert(ret == DW_DLV_OK);
    }

    assert(ret != DW_DLV_ERROR);
}

void dump_variables(const char *filename) {
    /* まず、 ELF 情報を取得する */
    int f;
    Elf_Cmd cmd;
    Elf *elf;
```

```
    elf_version(EV_CURRENT);

    f = open(filename, O_RDONLY);
    assert(f != -1);

    cmd = ELF_C_READ;
    elf = elf_begin(f, cmd, (Elf *) 0);
    process_one_file(elf, filename);
    elf_end(elf);
}

int main(int argc, char *argv[]) {
    dump_variables(argv[0]);
    return 0;
}
```

mainではdump_variables関数にargv[0]を渡しています。dump_variables関数では、ELF情報をlibelfを用いて取得しています。ここでは.aファイルの処理や、64bit ELF対応、Cygwin対応などを省略しています。詳しくはdwarfdumpのコードなどを参照して下さい。

次のprocess_one_file関数では、ELF情報からDWARF2情報を調べます。ここではdwarf_next_cu_headerを用いて各コンパイル単位をイテレートしています。そして得られたDwarf_Die型の変数dieをprocess_one_dieに渡しています。dieはDebug Information Entryの略で、デバッグ情報ツリーの各ノードに対応しています。

process_one_dieはdieを受け取り、それが変数であればその変数名と宣言行数を表示します。また、dieの兄弟をイテレートし、dieの子に対してprocess_one_dieを再帰的に呼び出します。ソースコード中では、next:以前が変数名表示処理、next:以降はツリーをたどる部分です。nextラベルは、特に問題のないエラーが発生した場合にcontinueの代わりにgoto next;を用いるために用意したものです。

サンプルの実行結果

サンプルは、以下のようにしてコンパイルして下さい。

```
% gcc -g dwarf.c -lelf -ldwarf
```

-gをつけてデバッグ情報を付加しないと何も出力されないことに気を付けて下さい。以下にサンプルの実行結果を示します。

```
10: dbg
10: die
10: d
11: err
12: ret
```

```
15: tag
16: child
24: attr
25: line
26: str
56: elf
56: filename
59: dbg
60: die
61: err
62: ret
64: cu_header_length
65: abbrev_offset
66: version_stamp
67: address_size
68: next_cu_offset
94: filename
96: f
97: cmd
98: elf
111: argc
111: argv
```

正しい変数一覧になっていることがわかります。

まとめ

GCC環境などで使用されるデバッグ情報であるDWARF2を扱うライブラリであるlibdwarfを紹介しました。

—— Shinichiro Hamaji

HACK #71 dumperで構造体のデータを見やすくダンプする

[Hack #70]を応用して作成されたdumperというライブラリを紹介します。

本Hackでは、「**[Hack #70] libdwarfでデバッグ情報を取得する**」を応用して作成されたdumperというライブラリの使用法と実装法を紹介します。dumperによってprintfデバッグやロギングなどをしたい時に簡単には表示しにくい構造体を表示できます。

dumperの使用法

dumperはhttp://shinh.skr.jp/binary/dumper.tgzで配布されており、ライセンスはLGPLとなっています。C++を用いたCから使用できるライブラリで、コンパイルにはlibelfとlibdwarfが必要です。

dumperは非常に簡単に使用することができます。基本的にはヘッダをインクルードしてdump_openを呼び出し、後はpというマクロに変数を渡せばその変数をダンプすることができます。以下に簡単なサンプルを示します。

```
#include "dump.h"

#include <string.h>

typedef enum { ENUM1, ENUM2 } TestEnum;
typedef union {
    int i;
    char b[4];
} TestUnion;

typedef struct TestDump_ {
    short s;
    int i;
    long l;
    unsigned long long ll;
    char c;
    char *str;
    void *ptr;
    void *const volatile *cvptr;
    struct TestDump_ *dump;
    int (*fp) (int, char*[]);
    int (*ifp) (int, char*[]);
    int array[10];
    TestEnum en;
    TestUnion un;
    struct {
    } no_name_struct;
} TestDump;

int main(int argc, char *argv[]) {
    TestDump d;
    d.s = 2;
    d.i = 3;
    d.l = 4;
    d.ll = 0xfffffffffffll;
    d.c = 'c';
    d.str = "hoge-";
    d.ptr = &d;
    d.cvptr = &d.ptr;
    d.fp = main;
    d.en = ENUM2;
    d.array[0] = 1;
    d.dump = &d;
    strcpy(d.un.b, "abc");

    dump_open(argv[0]);
```

```
    p(d);

    return 0;
}
```

このコードは、dump.oのあるディレクトリで、以下のようにコンパイルを行ってください。dump.cc が C++ で書かれているので、リンクには g++ を用いています。

```
% gcc -c -g dump_sample.c
% g++ dump_sample.o dump.o -lelf -ldwarf
```

TestDumpは構造体のそれぞれのメンバがどのようにダンプされるかを見るための構造体です。上記コードの実行結果は以下のようになります。

```
d = {
  s = 2 (0x0002) : short int
  i = 3 (0x00000003) : int
  l = 4 (0x00000004) : long int
  ll = 17592186044415 (0x00000fffffffffff) : long long unsigned int
  c = 'c' (63) : char
  str = "hoge-" [0x805e328] : char*
  ptr = 0xbfd42270 : void*
  cvptr = 0xbfd42270 [0xbfd4228c] : void**
  dump = 0xbfd42270 <previously shown> : TestDump_*
  fp = int main(int, char**) [0x8049bd4] : func*
  ifp = int ???(int, char**) [0x805e2ea] : func*
  array = { 1 (0x00000001), ... } : int[10]
  en = ENUM2 : TestEnum
  un = {
    i = 6513249 (0x00636261) : int
    b = "abc\x00" [0xbfd422cc] : char[4]
  } : TestUnion
  no_name_struct = {
  } : <no name>
} [0xbfd42270] : TestDump*
```

構造体にセットした内容が型情報とともに表示されていることがわかります。

dumperの実装

dumperは「**[Hack #70]　libdwarfでデバッグ情報を取得する**」の応用として実装されています。dump_openでは、指定したファイルを開いて、DWARF2情報を再帰的に調べて、すべての変数の名前、型、宣言されている行数、すべての型情報、関数シグネチャとアドレスを調べています。

また、pマクロで変数のダンプが行うこともできます。pマクロは以下のように宣言されて

います。

```
# define p(v)                                                \
    do {                                                     \
        typeof(v) *DUMP_TEMPVAL_NAME = &(v);                 \
        dump_s(&DUMP_TEMPVAL_NAME, __STRING(v), __FILE__, __LINE__); \
    } while(0)
```

まず最初に引数を対応する型のポインタに代入しています。マクロの引数は任意型なので、GCC拡張であるtypeofを用いて型名を調べています。そして次の行で、実際のダンプ関数に、ダンプしたい変数への参照、ダンプしたい変数の名前を__STRINGマクロを用いて文字列化したもの、__FILE__と__LINE__を用いた行情報、を渡しています。dump_sでは、行情報をさかのぼり、該当する変数宣言から型を調べ、その型情報に見合ったダンプを行います。

その他の応用

本Hackでは、デバッグ情報を残すことによって、Cなどでも実行時に十分な型情報を得られることがわかりました。このような方法を用いれば、シリアライザなどをCで実現することも可能でしょう。また、通常はソースコードを解析して行う、統合開発環境のインテリセンス補完などをデバッグ情報によって実装するのも面白いかもしれません。

ただし、デバッグ情報つきでコンパイルしなければならない、という点は大きな制限となることがあると考えられます。dumperは、デバッグ用途であればデバッグ情報必須であることは制限にならないだろう、という考えで作成しました。

また、その他の型情報を得るアプローチとして、GCC 4以降で-fdump-tree-generic-rawや-fdump-tree-gimple-rawオプションを用いて取得できるGENERICやGIMPLE（GCCの内部表現）をパースして取得する方法も考えられます。

—— Shinichiro Hamaji

HACK #72 オブジェクトファイルを自力でロードする

本Hackでは、オブジェクトファイルをロードする方法を紹介します。これにより、自分で作ったプラグイン機構の実現などが可能になります。

共有オブジェクトはリンカオプションで実行時にロードするほか、dlopen(3)によって実行時にロードすることもできます。本Hackでは、libbfdで得た情報をもとに.oを拡張子とするオブジェクトファイルを自力で再配置してロードする方法を紹介します。dlopen(3)については、「**[Hack #62] dlopenで実行時に動的リンクする**」を参照して下さい。

基本となるアイデア

「**[Hack #34]　ヒープ上に置いたコードを実行する**」では、mprotect(2)を用いてヒープのコードを実行する方法を示しました。ローダはまず、.oファイルや.aファイル（あまり意味はありませんが、.soファイルでも同じことはできます）をヒープにまるごとコピーしてしまうか、mmap(2)して、その部分にmprotect(2)で実行属性を付けてしまえばよいでしょう。また、ヒープにコピーする場合は必要なセクションだけコピーすればメモリの節約になるでしょう。

さらに、「**[Hack #67]　libbfd でシンボルの一覧を取得する**」で、すでにシンボルの名前とアドレスの一覧を取得する方法は紹介してあります。dlsym(3)相当のものはこの情報を用いて実装することができるでしょう。

残る問題は、メモリに置いた.oファイルや.aファイルは、まだ再配置が行われていないことです。本Hackでは、主としてこの点について議論します。

再配置情報を調べる

```
#include <stdio.h>

void hello() {
    puts("hello");
    puts("world!");
}
```

この非常に簡単なプログラムを、以下のようにコンパイルして、objdumpで情報を調べます。

```
% gcc -g -c hello.c
% objdump -Sr hello.o

hello.o:     ファイル形式 elf32-i386

セクション .text の逆アセンブル結果:

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

void hello() {
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 ec 08                sub    $0x8,%esp
    puts("hello");
   6:    83 ec 0c                sub    $0xc,%esp
   9:    68 00 00 00 00          push   $0x0
            a: R_386_32    .rodata
   e:    e8 fc ff ff ff          call   f <hello+0xf>
            f: R_386_PC32    puts
  13:    83 c4 10                add    $0x10,%esp
```

```
    puts("world!");
  16:    83 ec 0c                sub    $0xc,%esp
  19:    68 06 00 00 00          push   $0x6
            1a: R_386_32    .rodata
  1e:    e8 fc ff ff ff          call   1f <hello+0x1f>
            1f: R_386_PC32    puts
  23:    83 c4 10                add    $0x10,%esp
}
  26:    c9                      leave
  27:    c3                      ret
```

ここで、-Sはソースつきで逆アセンブルをするオプションで、-rは再配置情報を見るオプションです。これらを合わせて使用すると再配置情報が格段に見やすくなります。これを見ると、0x09アドレスにあるpush命令と0x0eアドレスにあるcall命令に再配置情報があり、まだ正しいアドレスがセットされておらず、それぞれ0x00000000と0xfffffffcが入っていることがわかります。順に見ていきましょう。

まず、.rodataの方ですが、これは引数の"hello"をpushしています。これは文字列定数ですので.rodata(read only dataセクション)にあることが考えられます。本当にあるのかを確認してみましょう。まずはobjdump -hで.rodataの位置を確認します。

```
% objdump -h hello.o

hello.o:     ファイル形式 elf32-i386

セクション:
索引名          サイズ      VMA       LMA       File off  Algn

... 中略 ...

  6 .rodata       0000000d  00000000  00000000  000001ee  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA

... 後略 ...
```

これを見ると.rodataの開始位置は0x1eeであることがわかります。「**[Hack #4] odでバイナリファイルをダンプする**」で紹介したodを用いてこの部分の内容を調べてみましょう。

```
% od -t x1z -j 0x1ee hello.o | head -1
0000756 68 65 6c 6c 6f 00 77 6f 72 6c 64 21 00 00 10 00  >hello.world!....<
```

helloとworld!が順に入っていることがわかります。odの-jオプションは指定したバイト数だけスキップするオプションです。1つ目のpushでは0x0が、2つ目のpushではもともと0x6という数字が入っていました。これらは、.rodataの先頭アドレスからのオフセットを表しています。結論として、R_386_32と指定されたアドレスを再配置する場合、元の数値に指

定されたセクションのアドレスを加算すればよいことがわかります。

次はcallの方の再配置ですが、こちらは一度実行ファイルにリンクして再配置が行われた出力を読んでみた方がよいでしょう。objdump -S で該当個所を調べると、以下のようになっています。

```
    puts("hello");
 80483aa:       83 ec 0c                sub    $0xc,%esp
 80483ad:       68 7c 84 04 08          push   $0x804847c
 80483b2:       e8 f1 fe ff ff          call   80482a8 <puts@plt>
 80483b7:       83 c4 10                add    $0x10,%esp
    puts("world!");
 80483ba:       83 ec 0c                sub    $0xc,%esp
 80483bd:       68 82 84 04 08          push   $0x8048482
 80483c2:       e8 e1 fe ff ff          call   80482a8 <puts@plt>
 80483c7:       83 c4 10                add    $0x10,%esp
}
```

右側の出力にある通り、0x080482a8 に puts@plt は存在しています。同じ関数を二度呼んでいますが、指定されている数値は0xfffffef1(-271)と0xfffffee1(-287)で、異なっています。もうおわかりかと思いますが、これは相対アドレス指定のcallだからです。callは次の命令の開始位置からの相対アドレスを調べるため、0x80483b7-271 = 0x80483c7-287 = 0x080482a8 で、正しく puts@plt の位置を指していることになります。

結局、R_386_PC32 という再配置情報が見つかった場合は、元の値に指定されていた関数のアドレスを加算し、再配置情報のあったアドレスを減算すればよいことがわかります。今回の場合、もともと0xfffffffc(-4) が入っていましたが、これはx86の相対callが次の命令からの相対アドレス指定であることからきています。

実際に実装する

ここまで来れば、実際の実装は(いくぶん面倒ではありますが)さほど難しくありません。前出の hello.o をロードするプログラムを示します。

```
#define _GNU_SOURCE

#include <bfd.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include <unistd.h>
#include <sys/mman.h>
#include <dlfcn.h>
```

```
bfd *abfd;
asymbol **syms;

/* hello シンボルを探し出す */
int get_hello_pos() {
    long storage;
    int symnum;
    int i;

    int hello_pos;

    storage = bfd_get_symtab_upper_bound(abfd);
    assert(storage >= 0);
    if (storage) syms = (asymbol**)malloc(storage);

    symnum = bfd_canonicalize_symtab(abfd, syms);
    assert(symnum >= 0);

    for (i = 0; i < symnum; i++) {
        asymbol *sym = syms[i];
        const char *name = bfd_asymbol_name(sym);
        if (strcmp(name, "hello") == 0) {
            /* これが hello シンボルの hello.o 内での位置 */
            hello_pos = abfd->origin + sym->section->filepos;
            break;
        }
    }

    return hello_pos;
}

unsigned char *load_hello_o(char *filename) {
    FILE *fp;
    int size = 0;
    unsigned char *hello_o;

    fp = fopen(filename, "rb");
    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    fseek(fp, 0, SEEK_SET);
    hello_o = (unsigned char *)malloc(size);
    fread(hello_o, 1, size, fp);
    fclose(fp);

    return hello_o;
}

void reloc_hello_o(unsigned char *hello_o) {
    asection *sect;
    arelent **loc;
    int size;
    int i;
```

```
    /* 実行コードだけ再配置する */
    sect = bfd_get_section_by_name(abfd, ".text");

    size = bfd_get_reloc_upper_bound(abfd, sect);
    assert(size >= 0);

    loc = (arelent **)malloc(size);

    size = bfd_canonicalize_reloc(abfd, sect, loc, syms);
    assert(size >= 0);

    for (i = 0; i < size; i++) {
        arelent *rel = loc[i];
        int *p = (int *)(hello_o + sect->filepos + rel->address);
        asymbol *sym = *rel->sym_ptr_ptr;
        const char *name = sym->name;

        /* セクションを再配置 */
        if ((sym->flags & BSF_SECTION_SYM) != 0) {
            asection *s = bfd_get_section_by_name(abfd, name);
            *p += (int)hello_o + s->filepos;
        }
        /* 関数を再配置 */
        else {
            /* hello.o の hello が hello.o 内の他の関数を呼んでいる場合は、
               その再配置先は dlsym ではなく syms の中から取得する必要がある */
            *p += (int)dlsym(RTLD_DEFAULT, name);
            if (rel->howto->pc_relative) *p -= (int)p;
        }
    }

    free(loc);
}

void invoke_hello(unsigned char *hello_o, int hello_pos) {
    void (*hello_fp) () = (void (*) ())(hello_o + hello_pos);

    /* メモリ保護を外している。
       「[Hack #34] ヒープ上に置いたコードを実行する」を参照 */
    int pagesize = (int)sysconf(_SC_PAGESIZE);
    char *p = (char *)((long)hello_fp & ~(pagesize - 1L));
    mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);

    hello_fp();
}

int main() {
    int ret;
    int hello_pos, hello_size;
    unsigned char *hello_o;

    char *filename = "hello.o";
```

```
    /* hello.o を開いて下準備 */
    abfd = bfd_openr(filename, NULL);
    assert(abfd);
    ret = bfd_check_format(abfd, bfd_object);
    assert(ret);

    if (!(bfd_get_file_flags(abfd) & HAS_SYMS)) {
        assert(bfd_get_error() == bfd_error_no_error);
        /* there are no symbols */
        bfd_close(abfd);
        return 1;
    }

    hello_pos = get_hello_pos();

    hello_o = load_hello_o(filename);

    reloc_hello_o(hello_o);

    free(syms);
    bfd_close(abfd);

    invoke_hello(hello_o, hello_pos);

    free(hello_o);
}
```

少し長いですが、順に見ていきましょう。まずmainの最初の部分はbfdの初期化部分です。また、get_hello_pos はシンボル一覧から hello というシンボルを探しています。この部分は「**[Hack #67] libbfd でシンボルの一覧を取得する**」とほとんど同じなのでそちらを参照して下さい。load_hello_o では hello.o を読み込み、単純にヒープにコピーしています。

reloc_hello_o がこの Hack のメインの関数です。まず、ここではデバッグ情報セクションなどは再配置しなくてもよいので、実行コードのある.textセクションを取得しています。そして、bfd_get_reloc_upper_boundとbfd_canonicalize_relocで再配置情報の一覧を取得します。この手続きはシンボル一覧を取得する場合と似ていると思います。注意すべき点は、bfd_canonicalize_reloc は、第4引数としてbfd_canonicalize_symtab で取得したsyms を必要とすることです。これが必要な事情ははっきりしないのですが、libbfd のドキュメント（http://www.sra.co.jp/wingnut/bfd/bfd-ja_2.html#SEC21）によると、「テーブル syms もまた、恐るべき内部的な神秘的理由により必要である（The SYMS table is also needed for horrible internal magic reasons）」とのことなので、ここでは深追いは避けておきます。

さて、次に取得した再配置情報を順番に見て、実際の再配置処理を行っていきます。まず hello.o 内の .text セクションの位置と rel->address から再配置の対象となるアドレスを計算しています。次に rel->sym_ptr_ptr からシンボル情報を取得し、シンボル名を調べ、シンボルのタイプによって条件分岐します。

シンボルがセクションであった場合は、該当セクションをbfd構造体から取得し、その位置を加算して再配置します。

シンボルが関数であった場合は、dlsym(3)を用いて関数のアドレスを調べ、その位置を加算して再配置します。さらに、rel->howto->pc_relativeが真である場合、これは相対アドレス指定の関数呼び出しですので、自分自身のアドレスを減算して相対にします。今回はhello関数は標準Cライブラリの関数のみ(puts)を呼び出しているため、dlsymで関数のアドレスを取得できましたが、hello.oの別の関数を呼び出す場合は、symsの情報からアドレスを調べる必要もあるでしょう。

ここまでですべての準備が完了しました。最後にinvoke_helloでhello関数のメモリ保護を外して実行しています。

補足

今回のHackでは、hello.oの中の、標準ライブラリの関数以外は呼び出していないhello関数のみを実行するために再配置を行いました。実際の再配置はもっと面倒な処理となります。例えば前述したロードされたオブジェクトファイル内での別な関数呼び出しや、.aファイルの読み込みなどはサポートされていません。また、再配置の種類はR_386_32とR_386_PC32だけではありません。いずれにせよ不十分ではありますが、筆者の作成したDTR（http://shinh.skr.jp/binary/dtr.html）ではもう少し丁寧な再配置が行われています。

今回のHackによって、Javaの.classファイルをロードするように、特にコンパイルオプションを変更することなくオブジェクトファイルをロードすることが可能になります。また、共有ライブラリがサポートされていない環境でプラグインを実現するためにも、今回のHackは使用できます。

XFree86では、プラグインのファイルは.aファイルとなっています(libdri.aなど)。これはまさに本Hackのような自前ローダによって実現されており、実際、Xのソースコードには各環境のオブジェクトファイルのローダが添付されていますので(hw/xfree86/loader/以下)、実装の参考になるでしょう。

まとめ

本Hackでは、オブジェクトファイルをロードする方法を紹介しました。このテクニックは、自前のプラグイン機構の実現などに使用することができます。

—— Shinichiro Hamaji

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 #74 GNU lightning でポータブルに実行時コード生成する

GNU lightningを用いると、ポータブルなアセンブラコードから機械語を実行時に生成することができます。

「**[Hack #34] ヒープ上に置いたコードを実行する**」では、ヒープに置いたコードを実行するためにメモリ保護を外す方法を紹介しました。この方法を用いれば実行時にネイティブコードを生成して実行できるのですが、どうしてもプロセッサ依存となってしまいます。本 Hack では GNU lightning というライブラリでプロセッサ非依存でこれを行う方法を紹介します。

GNU lightning

GNU lightning（http://www.gnu.org/software/lightning/）は実行時にアセンブラをポータブルに書くためのライブラリで、LGPL のもとで配布されています。同じようなコンセプトのライブラリとして、libjit（http://www.southern-storm.com.au/libjit.html）というライブラリもありますが、ここでは紹介しません。

GNU lightning を用いたプログラミングでは、C のコード上でプロセッサに依存しない抽象的なアセンブラを記述します。GNU lightning はこの抽象的なアセンブラを CPU に合わせて実行コードに変換してくれます。GNU lightning は x86、SPARC、PowerPC に対応しており、浮動小数点の機能は x86 のみが対象となっています。

GNU lightning を利用したプログラミングは、通常のアセンブラプログラミングに似ていますが、2つの点で大きく異なります。1つ目は、すべての命令が抽象化されているため、アーキテクチャのクセなどがあまりなく、特に関数呼び出しが抽象化されているため呼び出し規約などを意識しなくて良いことがあります。2つ目は、これも抽象化のために、レジスタの数は少ないアーキテクチャにあわせなければならないため、非常に少なく（6個）なっている

ことです。

C言語でポータブルなcurry化

ここではGNU lightningを用いてC言語で簡単なcurry化をポータブルに実装してみます。curry化とは、関数の引数の一部だけを先にセットしておいて引数の個数の減った関数を得る方法、といったようなものです。C++のSTLを知っている方は、bind1st、bind2ndを思い浮かべていただければ良いでしょう。以下に実装したコードを示します。

```
#include <stdio.h>

#include <lightning.h>

typedef int (*pifi)(int);
static jit_insn buf[1024];

pifi curry(int (*fp)(int, int), int a) {
    pifi code = (pifi)(jit_set_ip(buf).iptr);
    int i;

    jit_prolog(1);
    i = jit_arg_ui();
    jit_getarg_ui(JIT_V0, i);
    jit_movi_ui(JIT_V1, a);
    jit_prepare(2);
    jit_pusharg_ui(JIT_V1);
    jit_pusharg_ui(JIT_V0);
    jit_finish(fp);
    jit_retval(JIT_RET);
    jit_ret();
    jit_flush_code(buf, jit_get_ip().ptr);

    return code;
}

int add(int x, int y) {
    return(x + y);
}

int main() {
    pifi c;

    c = curry(add, 100);
    printf("%d\n", c(10));
}
```

mainの中で、addという加算を行う2引数関数と100という数値をcurry化して1引数関数を生成していることがわかります。この結果得られた関数を10という引数で実行しているた

め、100+10 が計算されて、110 が出力されます。では、肝心の curry 関数を順に見てみましょう。

まず、jit_prolog(1)で引数 1 つの関数を定義することを示しています。次に jit_arg_ui() と jit_getarg_ui で引数を取得します。引数の取得先は JIT_V0 という GNU lightning 形式のレジスタになっています。

次に、jit_movi_ui(JIT_V1, a)で curry の第 2 引数を JIT_V1 レジスタに渡しています。movi の i は即値(immediate) を表しており、_ui の ui は unsigned int を表しています。この段階はまだコード生成の段階であるため、a の値は即値として渡すことに注意して下さい。

必要な値はそろったので関数呼び出しに入ります。jit_pusharg_uiで2つの値を引数としてセットします。そして、jit_finish(fp)で関数を呼び出します。呼び出した関数の返り値は専用レジスタJIT_RETに入ります。jit_retval(JIT_RET)で、それをそのまま生成した関数の戻り値として、jit_ret()で関数を終了します。

最後に jit_flush_code でこれまでに書いたコードを buf 上に書き出します。この時 mprotect(2)によるメモリ保護外しも自動的に行われるため、あとは関数を実行するだけとなります。

GNU lightning の仕様

以上は非常に簡単なサンプルでしたが、GNU lightning は他にも多くのアセンブラの基本的な演算をサポートしています。例えば加減乗除や論理演算、条件分岐などです。これらの命令も、基本的に jit_movi_ui のように、jit_<op><immediate or register>_<type> という規則に従っています。例えばfloat レジスタにレジスタの値を加算する場合は、jit_addr_f というようになります。

使用できる int 型レジスタは JIT_V0、JIT_V1、JIT_V2、JIT_R0、JIT_R1、JIT_R2、JIT_RET の 7個となっています。JIT_V系は関数を越えて値が保存されることが保障されており、JIT_R系は関数呼び出しなどで値が変化してしまうかもしれません。JIT_RETは返り値を入れるためのレジスタです。JIT_RETに値を代入すると、アーキテクチャによっては他の汎用レジスタの値を破壊してしまう場合がありますから(例えばx86 では JIT_RET と JIT_R0 は両方とも eax レジスタです)、返り値の受け取りにだけ使用するのがよいでしょう。

また、GNU lightningには生成したコードをディスアセンブルする補助関数である、disassembleも付属しています。./configure --enable-disassemblingを実行することによってopcode/libdisass.a が作成されます。

まとめ

GNU lightning を用いると、ポータブルなアセンブラコードから機械語を実行時に生成す

ることができます。

—— Shinichiro Hamaji

HACK #75 スタック領域のアドレスを取得する

本Hackでは主なOSでプロセススタックとスレッドスタックのメモリアドレスを取得する方法を紹介します。

スタックを積極的に操作するガベージコレクションを実装する場合や、スタックオーバーフローを自前で捕捉しようとすると、スタック領域がメモリ空間のどの位置を占めているか調べる必要があります。本Hackでは動作中のスレッドからスタックのアドレス情報を取得する方法を紹介します。

マルチスレッドを使わない場合のスタック情報

UNIX系OSの(マルチスレッド化されていない)プロセスのスタックは、プロセスのメモリ空間の使い方に依存します。スタックのサイズは一般にgetrlimit(2)で取得できます。

```
struct rlimit rlim;
getrlimit(RLIMIT_STACK, &rlim);
size_t process_stack_size = (size_t)rlim.rlim_cur;
```

ただスタックの開始アドレスを知る標準的な方法がありません。決めうちの固定アドレスから始まるOSも多いのですが、いくつかのOSに関してプログラム的にスタック開始アドレスを得る方法を紹介します。

Linuxの場合

Linuxはプラットフォームごとにスタック開始アドレスの固定値があります(Linux/i686なら0xbfffffff)。しかしカーネル2.6.12から、スタック開始アドレスをランダムにずらすスタック保護機能がデフォルトになりました(/proc/sys/kernel/randomize_va_spaceで制御。固定値から最大8MBほど、スタックの成長方向にずれる)。

これを吸収してスタック開始アドレスを求めるには、glibc内のシンボル__libc_stack_endを用います。

```
#include <unistd.h> /* for sysconf(3) */
#include <stdint.h> /* for uintptr_t  */

#pragma weak  __libc_stack_end
extern void*  __libc_stack_end;

void* get_linux_stack_base() {
```

```
  long pagesize = sysconf(_SC_PAGESIZE);
  return (void *)(((uintptr_t)__libc_stack_end + pagesize) & ~(pagesize -1));
}
```

FreeBSD の場合

FreeBSD のスタック開始アドレスは sysctl で得ることができます。

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#include <assert.h>

void* get_freebsd_stack_base() {
  int nm[2] = {CTL_KERN, KERN_USRSTACK};
  void* base;
  size_t len = sizeof(void*);
  int r = sysctl(nm, 2, &base, &len, NULL, 0);
  assert(r);
  return base;
}
```

procfs が使える場合

「**[Hack #65]　ロードしている共有ライブラリをチェックする**」で紹介されている手法でスタック領域を探し当てることができます。ローカル変数のアドレスからスタックポインタ(SP)を求めておき、SPアドレスを含む領域を探します。一部のOSはprocfsと同様の情報をAPI経由で得られます。HP-UX では pstat(2)が使用可能です。

```
#include <sys/param.h>
#include <sys/pstat.h>

struct pst_vm_status vm_status;
int i = 0;
while (pstat_getprocvm(&vm_status, sizeof(vm_status), 0, i++) == 1) {
  if (vm_status.pst_type == PS_RSESTACK) {
    void* base = (void*)vm_status.pst_vaddr;
    size_t size = (size_t)vm_status.pst_length;
  }
}
```

その他

次に紹介するスレッドスタック情報の取得するAPIで、プロセススタックの情報が取れるOSがあります。スレッドライブラリをアタッチするのも1つの手です。

マルチスレッドのスレッドスタックは？

プロセスの中で新たに生成されたスレッドは、スレッドライブラリが自動的にスタック領域を割り当てます。残念ながらこのスレッドスタック領域のアドレスを取得する標準的な方法は用意されていません。いくつかのOSは独自のAPIを用意しています(表5-1)。

表5-1 各OSごとのスタック領域情報の取得API

OS	API名	インクルードファイル
Linux	pthread_getattr_np	pthread.h
FreeBSD	pthread_attr_get_np	pthread_np.h
OpenBSD	pthread_stackseg_np(3)	pthread_np.h & sys/signal.h
Mac OS X	pthread_get_stacksize_np, pthread_get_stackaddr_np	pthread.h
Solaris	thr_stksegment(3thr)	thread.h & sys/signal.h

上の表中の_npというサフィックスはnon-portableを意味します。移植性のないAPIなので注意が必要です。

Linuxの場合のHack

Linuxはpthread_getattr_npを用いることで生成後のスレッドから属性情報を取得することができますが、スレッドやライブラリの種類によってはpthread_getattr_npが定義されていなかったり、正しい情報を返さないことがありますので条件で分ける必要があります。

1. 最初から存在するプロセスのスレッドにはpthread_getattr_npは正しい情報を返しません。プロセスのメインスレッドにはgetrlimitからの情報を使いましょう。
2. 古いスレッドライブラリにはpthread_getattr_npが存在しません。このような古いライブラリでは、スレッドスタックは2MB境界に沿った2MBに固定されています。スタックポインタの位置からスタック領域を逆算可能です。

これらを踏まえるとLinuxのスレッド領域を返すget_linux_stack_infoは以下のコードになります。

```
#include <pthread.h>
#include <sys/resource.h>
#include <unistd.h>
#include <dlfcn.h>
#include <unistd.h>
#include <stdint.h>
```

```
#pragma weak  __libc_stack_end
extern  void* __libc_stack_end;

#define INITIAL_PROCESS_STACK_END ((char*)0xC0000000U)
#define DEFAULT_FIXED_STACK_SIZE (2 * 1024 * 1024)

typedef int       (*GETATTR_NP_FUNC)(pthread_t, pthread_attr_t *);
typedef int       (*ATTR_GETSTACKBASE_FUNC)(pthread_attr_t *, void**);
typedef int       (*ATTR_GETSTACKSIZE_FUNC)(pthread_attr_t *, size_t*);
typedef int       (*ATTR_GETSTACK_FUNC)(pthread_attr_t *, void** stackaddr, size_t*
stacksize);

int get_linux_stack_info(void** stackaddr, size_t* stacksize) {
  char dummy;
  char* p = &dummy;
  char* initial_process_stack_end = INITIAL_PROCESS_STACK_END;
  size_t process_stack_size = 0;
  long pagesize = sysconf(_SC_PAGESIZE);
  struct rlimit rlim;

  getrlimit(RLIMIT_STACK, &rlim);
  process_stack_size = (size_t)rlim.rlim_cur ;

  if (&__libc_stack_end && __libc_stack_end) {
    initial_process_stack_end = (char*)(((uintptr_t)__libc_stack_end + pagesize) &
~(pagesize -1));
  }

  if (initial_process_stack_end - process_stack_size <= p &&
      p <=  initial_process_stack_end) {
    /* プロセススレッドの場合 */
    *stackaddr = (void*)(initial_process_stack_end - process_stack_size);
    *stacksize = process_stack_size;
    return 0;
  } else {
    GETATTR_NP_FUNC getattr_np_func =
      (GETATTR_NP_FUNC)dlsym(NULL, "pthread_getattr_np");

    if (!getattr_np_func) {
      /* 古いスレッドシステムではスタックサイズは 2MB 固定 */
      *stackaddr = (void*)( (size_t)p & (DEFAULT_FIXED_STACK_SIZE - 1));
      *stacksize = DEFAULT_FIXED_STACK_SIZE;
      return 0;
    } else {
      pthread_attr_t attr;
      pthread_attr_init(&attr);

      if (!getattr_np_func(pthread_self(), &attr)) {
        ATTR_GETSTACK_FUNC attr_getstack_func =
          (ATTR_GETSTACK_FUNC) dlsym(NULL, "pthread_attr_getstack" );
        ATTR_GETSTACKBASE_FUNC  attr_getstackaddr_func =
          (ATTR_GETSTACKBASE_FUNC) dlsym(NULL, "pthread_attr_getstackaddr");
```

```
        ATTR_GETSTACKSIZE_FUNC  attr_getstacksize_func =
          (ATTR_GETSTACKSIZE_FUNC) dlsym(NULL, "pthread_attr_getstacksize");

        if (attr_getstack_func) {
          int ret = attr_getstack_func(&attr, stackaddr, stacksize);
          pthread_attr_destroy(&attr);
          return ret;
        } else if (attr_getstackaddr_func && attr_getstacksize_func) {
          int ret = attr_getstackaddr_func(&attr, stackaddr) ||
                    attr_getstacksize_func(&attr, stacksize);
          pthread_attr_destroy(&attr);
          return ret;
        }
      }
      pthread_attr_destroy(&attr);
    }
  }

  return -1;
}
```

Windowsの場合のHack

Windowsの場合はThread Information Block（TIB）にスレッドに関する情報が記憶されています。x86/Windowsの場合にセグメントレジスタFSがスレッドごとに異なるセグメントを指すようにセットされており、以下のコードでTIBを取得可能です。

```
/* TIB を取得する関数 */
NT_TIB* getTIB(void) {
  NT_TIB* pTib;
  __asm {
    mov eax, dword ptr FS:[18H];
    mov pTib, eax;
  }
  return pTib;
}

/* スタック領域の判定 */
NT_TIB* pTIB = getTIB();
printf("[%p %p]\n", pTIB->StackBase, pTIB->StackLimit);
```

NT_TIBはプラットフォームSDKのWinNT.hを参考にしてください。

まとめ

本ハックでは主なOSでプロセススタックとスレッドスタックのメモリアドレスを取得する方法を紹介しました。スタック領域の調査方法はOSのバージョンが異なると動かなくなることがあります。実際に使用する前にターゲット環境で動作するかどうかのテストが欠か

せません。

—— Minoru Nakamura

HACK #76 sigaltstackでスタックオーバーフローに対処する

本Hackではsigaltstackを使い、スタックオーバーフローをハンドリングする手法を紹介します。コンパイラやインタプリタランタイムなどどこまでスタックを使うか分からないプログラムでは、スタックオーバーフローのケアを忘れないようにしましょう。

再帰呼び出しやalloca(3)を積極的に使うようなプログラムでは、スタックが上限を超えてしまいスタックオーバーフローが発生することがあります。本 Hack ではスタックオーバーフローエラーを捕捉して、適切にリカバリーする方法を紹介します。

OS がスタックオーバーフローを検出する仕組み

スタック領域がプロセッサ空間のどこに配置されるかはスレッドが生成された時に決定されます。ほとんどの UNIX では図 5-1 のように高位のアドレスからスタックを使い始め、下方向(低位アドレス)に伸ばして行きます。スタックオーバーフローの検出は、スタック領域の終端となる最低位の仮想記憶ページ(数ページのこともある)のページ属性をPROT_NONEとすることで、アクセス禁止にすることで実現します。スタックが伸びてきてこの領域にアクセスすると、メモリ保護が働きSEGVシグナル(SIGSEGV)が発生することになるのです。この禁止領域は「ガードページ(guard page)」あるいは「レッドゾーン(red zone)」と呼ばれます。

PA-RISCとIA-64では特殊なスタック配置が必要になります。PA-RISCのスタックは低位アドレスから始まり高位アドレスに伸びて行くため、レッドゾーンは最も高位のアドレスに配置されます。IA-64は通常のスレッドスタックとは別にレジスタ、スタックの退避先が必要な

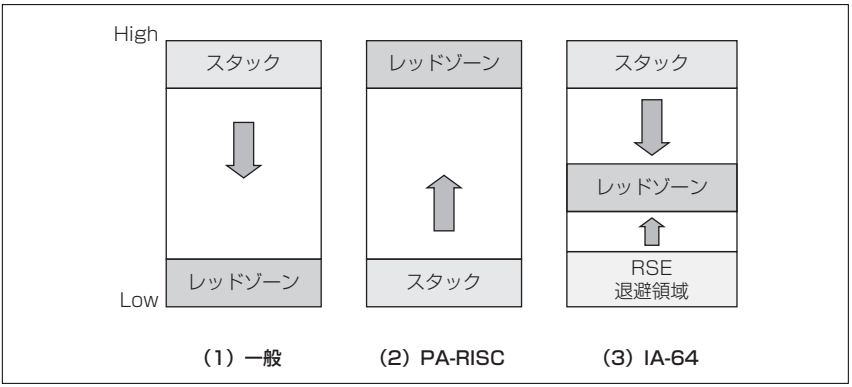


図 5-1　スタック領域のイメージ

ため、真中にレッドゾーンを配置し、高位から低位はスレッドスタックとしてプログラムが、低位から高位はレジスタスタックエンジンが使用することになっています。

sigaltstackを使う

スタックオーバーフローのSEGVシグナルはsignal(2)やsigaction(2)だけでは補捉できません。SEGVなどの同期シグナルは元のスレッドのスタックを利用するため、スタックオーバーフロー後はシグナルハンドラを動かすスタックもないのです。そのためスタックオーバーフローを補捉するためには「代替シグナルスタック(alternate signal stack)」を設定する必要があります。代替シグナルスタックを設定すると、シグナルハンドラではオリジナルのスタックスレッドの替わりに代替シグナルスタックが使用されます。sigaltstack(2)を使うことでスレッドごとに設定することが可能です。

代替シグナルスタックとする領域は、malloc(3)などで確保して渡します。以下のコードはスタックがレッドゾーンに達してSEGVシグナルが発生した時にsigsetjmp/siglongjmpを使って元のプログラムに復帰する例です。

サンプルコード1

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <setjmp.h>

#define ALT_STACK_SIZE (64*1024)

static sigjmp_buf return_point;

static void signal_handler(int sig, siginfo_t* sig_info, void* sig_data) {
  if (sig == SIGSEGV) {
    siglongjmp(return_point, 1);
  }
}

static void meaningless_recursive_func() {
  meaningless_recursive_func();
}

static void register_sigaltstack() {
  stack_t newSS, oldSS;

  newSS.ss_sp    = malloc(ALT_STACK_SIZE);
  newSS.ss_size  = ALT_STACK_SIZE;
  newSS.ss_flags = 0;

  sigaltstack(&newSS, &oldSS);
```

```
}

int main(int argc, char** argv) {
  struct sigaction newAct, oldAct;

  /* 代替シグナルスタックの設定 */
  register_sigaltstack();

  /* シグナルハンドラの設定 */
  sigemptyset(&newAct.sa_mask);
  sigaddset(&newAct.sa_mask, SIGSEGV);
  newAct.sa_sigaction = signal_handler;
  newAct.sa_flags = SA_SIGINFO|SA_RESTART|SA_ONSTACK;

  sigaction(SIGSEGV, &newAct, &oldAct);

  /* わざとスタックオーバーフローを発生させてハンドルする */
  if (sigsetjmp(return_point, 1) == 0) {
    meaningless_recursive_func();
  } else {
    fprintf(stderr, "stack overflow error\n");
  }

  return 0;
}
```

イエローゾーンを設ける

前述のレッドゾーンでのSEGVシグナルを直に補捉する方法はスタックオーバーフローを捕捉する基本ですが、実際のプログラムに応用するといろいろ問題にぶつかります。シグナルハンドラからsiglongjmp(3)で戻ってしまうと、スレッド内でオープンした資源の後始末ができないのです。

この問題を解決するためにイエローゾーンというテクニックがあります。スタック内の

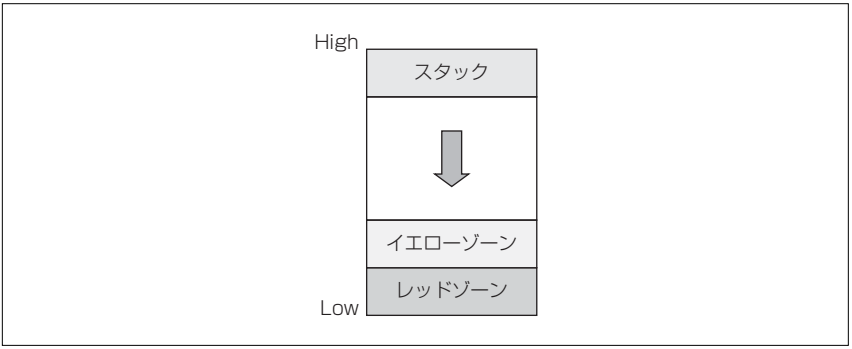


図 5-2　イエローゾーンを挟み込んだスタック領域のイメージ

レッドゾーンの手前にPROT_NONE属性を付けた仮想記憶ページをもう1ページ(あるいは数ページ)余分に設定し、これをSEGVシグナルで捕捉するという方法です。この余分なページをイエローゾーンと呼びます。イエローゾーンのSEGVシグナルが発生したら、シグナルハンドラの中でmmap(2)を使いメモリをマップし、mprotect(2)で書き込み、読み込み可能属性を設定して、元のコンテキストに復帰します。元に戻るとイエローゾーン分だけスタックが余分に使えるようになっています。

下のコードはLinuxでイエローゾーンを使う例です。i386/RHEL4、i386/RHEL3、Vine Linux 3.1などで動作を確認しています。この手法はスタック領域がメモリ空間上のどこに配置されているのか正確に知る必要がありますが、スタック領域情報を取得する方法はOSごとに書き方が異なります。「**[Hack #75] スタック領域のアドレスを取得する**」を参考にしてget_stack_info()を入れ替えてください。yellow_zone_hook()はイエローゾーンのSEGVシグナルが起きた時の処理を記述しますが、「**[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**」のようなHackを併用することで、元のコンテキスト側にC++の例外を発生させるなどの応用が可能です。

サンプルコード2

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <sys/mman.h>
#include <unistd.h>
#include <assert.h>
#include <sys/resource.h>

#define ALT_STACK_SIZE    (64*1024)
#define YELLOW_ZONE_PAGES (1)

/* スレッドごとのスタック情報。Thread-specific data で記録。
 * この構造体は downward-growing stack を想定し、3つのポインタはいずれもページ境界にあるものとする。
 *
 *  +---------------+ <- stack_pointer + stack_size
 *  |               |
 *  +---------------+ <- yellow_zone_boundary
 *  |               |
 *  +---------------+ <- red_zone_boundary
 *  |               |
 *  +---------------+ <- stack_pointer(スタックの最低位のアドレス)
 *
 */
typedef struct {
 size_t      stack_size;           /* スタック領域のサイズ                          */
 char*       stack_pointer;        /* スタックの最低位のアドレス                    */
 char*       red_zone_boundary;    /* レッドゾーンとの境界(レッドゾーン自体含まれず)*/
 char*       yellow_zone_boundary; /* イエローゾーンとの境界。
```

```
                                    /* イエローゾーンがない場合は NULL              */
  sigjmp_buf  return_point;         /* レッドゾーンを越えた場合の戻り先             */
  size_t      red_zone_size;        /* 作業用 */
} ThreadInfo;

/* ThreadInfo を記録するための Thread-specific key で記録。*/
static pthread_key_t     thread_info_key;

/* シグナルハンドラ */
static struct sigaction  newAct, oldAct;

/*************************************************************************/
/* ユーザー定義ルーチン                                                  */
/*************************************************************************/

/* メインの処理 */
static void main_routine() {
  /* 再帰呼び出しでわざとスタックオーバフローを発生させている */
  main_routine();
}

/* レッドゾーンに到達した時の処理 */
static void stackoverflow_routine() {
  fprintf(stderr, "stack overflow error.\n");
  fflush(stderr);
}

/* イエローゾーンを超過した場合の処理 */
static void yellow_zone_hook(/* シグナルハンドラから欲しい情報を引数でもらう */) {
  /* 好きな処理をここに記述 */
  fprintf(stderr, "exceeded yellow zone.\n");
  fflush(stderr);
}

/* スタック領域情報取得 (ここはプラットフォームごとに書き方が異なる) */
static int get_stack_info(void** stackaddr, size_t* stacksize) {
  int ret = -1;
  pthread_attr_t attr;
  pthread_attr_init (&attr);

  if (pthread_getattr_np(pthread_self(), &attr) == 0) {
    ret = pthread_attr_getstack(&attr, stackaddr, stacksize);
  }
  pthread_attr_destroy (&attr);

  return ret;
}

/*************************************************************************/
/* スタックオーバーフローハンドリングの骨格                              */
/*************************************************************************/
```

```
/* ポインタがスタック領域内にある */
static int is_in_stack(const ThreadInfo* tinfo, char* pointer) {
  return (tinfo->stack_pointer <= pointer) && (pointer < tinfo->stack_pointer + tinfo-
>stack_size);
}

/* ポインタがレッドゾーンの中にある */
static int is_in_red_zone(const ThreadInfo* tinfo, char* pointer) {
  assert(tinfo->red_zone_boundary);
  return (tinfo->stack_pointer <= pointer) && (pointer < tinfo->red_zone_boundary);
}

/* ポインタがイエローゾーンの中にある */
static int is_in_yellow_zone(const ThreadInfo* tinfo, char* pointer) {
  if (tinfo->yellow_zone_boundary) {
    return (tinfo->red_zone_boundary <= pointer) && (pointer < tinfo-
>yellow_zone_boundary);
  }
}

/* イエローゾーンをセットする */
static void set_yellow_zone(ThreadInfo* tinfo) {
  int pagesize = sysconf(_SC_PAGE_SIZE);
  assert(pagesize > 0);
  tinfo->yellow_zone_boundary = tinfo->red_zone_boundary + pagesize * YELLOW_ZONE_PAGES;
  mprotect(tinfo->red_zone_boundary, pagesize * YELLOW_ZONE_PAGES, PROT_NONE);
}

/* イエローゾーンを解除する */
static void reset_yellow_zone(ThreadInfo* tinfo) {
  size_t pagesize = tinfo->yellow_zone_boundary - tinfo->red_zone_boundary;
  if (mmap(tinfo->red_zone_boundary, pagesize, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0) == 0) {
    perror("mmap failed"), exit(1);
  }
  mprotect(tinfo->red_zone_boundary, pagesize, PROT_READ|PROT_WRITE);
  tinfo->yellow_zone_boundary = 0;
}

/* シグナルハンドラ */
static void signal_handler(int sig, siginfo_t* sig_info, void* sig_data) {
  if (sig == SIGSEGV) {
    ThreadInfo* tinfo = (ThreadInfo*)pthread_getspecific(thread_info_key);
    char* fault_address = (char*)sig_info->si_addr;

    if (is_in_stack(tinfo, fault_address)){
      if (is_in_red_zone(tinfo, fault_address)) {
        /* レッドゾーンに突入した場合 */
        siglongjmp(tinfo->return_point, 1 /* ここは 0 でない任意の整数 */ );
      } else if (is_in_yellow_zone(tinfo, fault_address)) {
        /* イエローゾーンに突入した場合 */

        /*
```

```
         * イエローゾーンを通常のスタックに戻すことで現在発生している SEGV は解除される。
         * これで猶予期間を稼ぐことができる。
         */
        reset_yellow_zone(tinfo);

        /*
         * 回避できないスタックオーバーフローが接近していることを
         * メインのプログラム側に伝達する。
         */
        yellow_zone_hook(/* 必要な情報を並べる */);
        return;
      } else {
        /* スタック領域内でオーバーフローとは無関係な SEGV が発生している */
      }
    }
    /* 必要に応じて */
    /* oldAct.sa_sigaction(sig, sig_info, sig_data); */
  }
}

/* アプリケーションで一度だけシグナルハンドラや TS キーを登録する */
static void register_application_info() {
  /* Thread-specific キーの登録 */
  pthread_key_create(&thread_info_key, NULL);

  /* SEGV シグナルハンドラをセット */
  sigemptyset(&newAct.sa_mask);
  sigaddset(&newAct.sa_mask, SIGSEGV);
  newAct.sa_sigaction = signal_handler;
  newAct.sa_flags     = SA_SIGINFO | SA_RESTART | SA_ONSTACK;

  sigaction(SIGSEGV, &newAct, &oldAct);
}

/* スレッドごとに TSD や大体シグナルハンドラを登録する */
static void register_thread_info(ThreadInfo* tinfo) {
  stack_t ss ;

  /* TSD へ tinfo を登録 */
  pthread_setspecific(thread_info_key, tinfo);

  /* スタック領域の登録 */
  get_stack_info((void**)&tinfo->stack_pointer, &tinfo->stack_size);

  /* レッドゾーンの登録 */
  tinfo->red_zone_boundary = tinfo->stack_pointer + tinfo->red_zone_size;

  /* イエローゾーンの登録 */
  set_yellow_zone(tinfo);

  /* 代替シグナルスタックの登録 */
  ss.ss_sp    = (char*)malloc(ALT_STACK_SIZE);
  ss.ss_size  = ALT_STACK_SIZE;
```

```
  ss.ss_flags = 0;
  sigaltstack(&ss, NULL);
}

/* 各スレッドのベース部分 */
static void* thread_routine(void* p) {
  ThreadInfo* tinfo = (ThreadInfo*)p;

  /* スレッドごとの情報登録 */
  register_thread_info(tinfo);

  if (sigsetjmp(tinfo->return_point, 1) == 0) {
    /*  メインの処理 (この中でスタックオーバーフローが発生する可能性) */
    main_routine();
  } else {
    /*  レッドゾーンからのリターンポイント */
    stackoverflow_routine();
  }
  free(tinfo);
  return 0;
}

int main(int argc, char** argv) {
  /* シグナルハンドラの登録 */
  register_application_info();

  if (argc == 2) {
    int stacksize = atoi(argv[1]);

    pthread_attr_t attr;
    pthread_attr_init (&attr);
    pthread_attr_setstacksize(&attr, 1024 * 1024 * stacksize);

    {
      pthread_t pid0;
      ThreadInfo* tinfo = (ThreadInfo*)calloc(1, sizeof(ThreadInfo));
      pthread_attr_getguardsize(&attr, &tinfo->red_zone_size);
      pthread_create(&pid0, &attr, thread_routine, tinfo);
      pthread_join(pid0, NULL);
    }
  } else {
    printf("Usage: %s stacksize(mb)\n", argv[0]);
  }
  return 0;
}
```

古いLinuxでsigaltstackを無理矢理使う

比較的古いLinuxはFixed stackと呼ばれる2MB固定のスタック領域を持っています。Fixed stackはスタック領域を2MB境界に沿ったメモリ空間上に配置し、この特性を利用してス

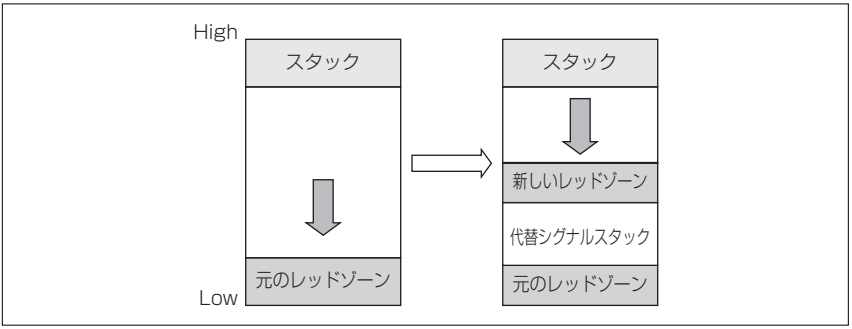


図 5-3　Fixed stack の Linux で sigaltstack を使う

レッドの識別やスタック領域の把握を行っています。malloc(3)など外側のメモリをsigaltstackに割り当てると、スレッドの識別がうまくいかず意図しないエラーが発生します。

Red Hat Linux 7.0 以前などで問題の起こすカーネル & スレッドライブラリが使用されています。これらのディストリビューションでsigaltstackを動作させるには少し特殊なHackが必要です。Fixed stackではスタックポインタが2MB領域の外側にあると障害が発生しますので、図5-3のように代替シグナルスタックを元のスタックの中に埋め込んでしまえばよいのです。サンプルコード1を元に、register_sigaltstack()を以下のように変更すればOKです。

サンプルコード3

```
#include <sys/mman.h>

#define ORIGINAL_RED_ZONE_SIZE (64*1024)   /* マージンを多めにとる */
#define NEW_RED_ZONE_SIZE      (4*1024)    /* x86 のページサイズ   */
#define FIXED_STACK_SIZE       (2*1024*1024)

static void register_sigaltstack() {
  stack_t newSS, oldSS;
  char* stack_base = (char*)((size_t)&newSS & ~(FIXED_STACK_SIZE-1));

  /* スタック領域内に代替シグナルスタックを設定 */
  newSS.ss_sp    = (void*)(stack_base + ORIGINAL_RED_ZONE_SIZE);
  newSS.ss_size  = ALT_STACK_SIZE;
  newSS.ss_flags = 0;

  sigaltstack(&newSS, &oldSS);

  /* 代替シグナル領域と新しいレッドゾーンにメモリをマップする */
  mmap(newSS.ss_sp, newSS.ss_size + NEW_RED_ZONE_SIZE,
       PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

  /* 新しいレッドゾーンを PROT_NONE に設定 */
  mprotect(stack_base + ORIGINAL_RED_ZONE_SIZE + ALT_STACK_SIZE,
```

```
                NEW_RED_ZONE_SIZE, PROT_NONE);
    }
```

注意事項

sigaltstackはバグや制限を抱えた実装系が多いようです。

- Solarisの場合、Solaris 8以前のスレッドライブラリではLWPにboundされないスレッドの中でsigaltstackが使用できません。
- 代替シグナルスタックがオーバーフローすることは想定されていません。あふれないように注意する必要があります。
- sigaltstackを設定したスレッドからpthread_createで子スレッドを作成すると、代替シグナルスタックの設定がコピーされる処理系が多いようです。この場合、親スレッドと子スレッドで同時にシグナルが発生すると意図しないメモリ破壊が起きる危険性があります。

まとめ

本Hackではsigaltstackを使い、スタックオーバーフローをハンドリングする手法を紹介しました。小さなプログラムではスタックオーバーフローは意識することはないと思いますが、コンパイラやインタプリタランタイムなどどこまでスタックを使うか見積もることができないプログラムでは、スタックオーバーフローのケアを忘れないようにしましょう。

—— Minoru Nakamura

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 #78 シグナルハンドラからプログラムのコンテキストを書き換える

シグナルハンドラから、中断された側の文脈（ここではプログラムカウンタ）を書き換えてみます。

シグナルでプログラムの実行が中断された際に、シグナルハンドラから中断された側のコンテキストを書き換える方法を紹介します。

コンテキスト書き換え

シグナルが発生するとプログラムの実行は中断され、制御はシグナルハンドラに移ります。シグナルハンドラの実行が終了すると、通常は、中断された個所からプログラムの実行が再開されるものです。本 Hack では、シグナルハンドラから、実行が中断されている側のプログラムに対してコンテキスト、つまり状態の書き換えを試みます。具体的には、文脈中のプログラムカウンタを書き換えることで、中断された個所とは異なる個所から実行を再開させます。

以下に、Linux/x86 用のプログラムを示します。

```
#include <stdio.h>
#include <signal.h>
#include <asm/ucontext.h>

static unsigned long target;

void handler(int signum, siginfo_t *siginfo, void *uc0) {
  struct ucontext *uc;
  struct sigcontext *sc;

  uc = (struct ucontext *)uc0;
  sc = &uc->uc_mcontext;

  sc->eip = target;
}

int main(int argc, char **argv) {
  struct sigaction act;

  act.sa_sigaction = handler;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGTRAP, &act, NULL);

  asm("movl $skipped,%0" : : "m" (target));

  asm("int3");      /* causes SIGTRAP */
```

```
    printf("To be skipped.\n");
    asm("skipped:");
    printf("Done.\n");
}
```

main()関数は、printf()を2回呼んでいます。何事もなければ、次の2行が表示されるはずです。

```
To be skipped.
Done.
```

しかし実行してみると、1行目は表示されずDone.だけが表示されます。以下、何が起きているのか、main()関数から見ていきます。

sigaction(2)を呼び出している行までで、シグナルハンドラを設定しています。ここでは、SIGTRAPという種類のシグナルが発生した場合にhandler()を呼び出す、という設定を行っています。続くインラインアセンブリコードasm("movl ...は、変数targetにラベルskippedのアドレスを代入しています。ラベルskippedはその少し先にasm("skipped:")と書かれています。

ここまでで準備は完了です。続くアセンブリコードint3は、SIGTRAPという種類のシグナルを発生させます。int3については「**[Hack #92] Cのプログラムの中でブレークポイントを設定する**」を参照してください。

SIGTRAPが発生すると、main()実行中のプログラムの状態、つまり文脈は保存され、制御はシグナルハンドラhandler()に移ります。handler()はその文脈(struct sigcontext型)を取得して、変数scにポインタを格納します。文脈の取得方法は、OSによりさまざまである点に注意してください。

この文脈を書き換えることで、シグナルハンドラから制御が戻った後のプログラムの状態を変更することができます。ここでは、文脈に含まれるプログラムカウンタを書き換えて、実行が再開される個所を変更します。

handler()は、最後の行でメンバeipの値を書き換えます。

```
sc->eip = target;
```

このeipはx86プロセッサのプログラムカウンタであり、targetは先に保存しておいたラベルskippedのアドレスです。つまり、通常であればprintf("To Be...から実行が再開されるところを、ラベルskippedの個所から再開されるように変更しています。

Linux以外の場合

Linuxでは、シグナルハンドラの第3引数がstruct ucontext *型で、文脈(struct sigcontext

* 型）はそのメンバとなっています。FreeBSD や NetBSD には、第 3 引数が直接文脈を指すバージョンもあります。また、文脈中でレジスタを表すメンバの名前も、Linuxではeipであるところが FreeBSD、NetBSD では sc_eip となっています。

まとめ

シグナルハンドラから、中断された側の文脈、ここではプログラムカウンタを書き換えました。使い途が限られる技法ですが、言語実行系の例外処理などに使える手法ではあります。

—— Kazuyuki Shudo

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" (p));
```

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 #80 自己書き換えでプログラムの動作を変える

現代的なUNIX系OSの上で自分自身を書き換えるプログラムを紹介します。また、これを実現するために必要なテクニックや注意事項も解説します。

以前はメモリの節約といった目的でよく行われていた自己書き換えを、現代のUNIX系OSで実践します。

自分自身の挙動を書き換えるプログラム

自分自身のコードを書き換えて動作を変える関数を示します。Linux/x86用のコードです。

```
#include <stdio.h>
#include <unistd.h>     /* for sysconf(3) */
#include <sys/mman.h>   /* for mprotect(2) */

void func(void) {
  printf("1st.\n");

  asm("slot:\n\t"
      "nop\n\t"
      "nop\n\t"
      "after_slot:");

  printf("2nd.\n");
```

```
  /* allows code modification */
  long pagesize = (int)sysconf(_SC_PAGESIZE);
  char *p = (char *)((long)func & ~(pagesize - 1L));
  mprotect(p, pagesize * 10L, PROT_READ|PROT_WRITE|PROT_EXEC);

  /* modifies func() itself */
  asm(".byte 0xe8,0,0,0,0\n\t"  /* call the following popl */
      "popl %%eax\n\t"
      "addl $0x14, %%eax\n\t"
      "subl $after_slot, %%eax\n\t"
      "shl  $8, %%eax\n\t"
      "movb $0xeb, %%al\n\t"     /* EB is jmp (2 byte insn) */
      "movw %%ax, slot" : : : "eax");

  printf("3rd.\n");
}

int main(int argc, char **argv) {
  func();
  func();
}
```

main()関数がfunc()関数を2度呼び出しています。func()関数はprintf()を呼んでメッセージを表示します。インラインアセンブリコード(asm(..))がなければ、出力は次のようになるはずです。

```
1st.
2nd.
3rd.
1st.
2nd.
3rd.
```

しかし実際は、2度目の「2nd.」は表示されません。3回以上func()を呼び出したとしても、二度と「2nd.」は表示されません。

func()は、初回実行時に自分自身を書き換えます。この書き換えによって、2度目以降の実行ではprintf("2nd.\n")をスキップするようになります。

説明

func()の処理を順に見ていきます。1st.表示後のnop命令は、何もしないというx86命令です。後ほど、この2バイトのnopは他の命令で上書きされます。

2nd.を表示した後、mprotect(...)までの3行で、自分自身、つまりfunc()のコードを書き換えられるようにします。mprotect(2)については「**[Hack #34] ヒープ上に置いたコードを実行する**」を参照してください。

mprotect(...)に続くアセンブリコードが“肝”です。前述のnopの上に、2nd.の表示をスキップするようなジャンプ命令を書き込みます。これによって、その後のfunc()の実行では2nd.は表示されないようになります。

このアセンブリコードは、先頭の2命令で、このコード自身が置かれているメモリ領域のアドレスを取得します。

```
.byte 0xe8,0,0,0,0
popl  %%eax
```

この命令列は、EAXレジスタにpopl命令自身のアドレスを格納します。このことについては「**[Hack #79] プログラムカウンタの値を取得する**」を参照してください。

その後は、popl命令のアドレスを元にいろいろな操作をして、ラベルafter_slotから見たprint("3rd.\n")の相対アドレスを算出します。そして、次の2バイトを、前述のnop命令2バイトに上書きします。

```
0xeb この相対アドレス
```

0xebはジャンプ命令です。このジャンプの飛び先はprint("3rd.\n)となります。これで晴れて、nopがあった場所に実行がさしかかると3rd.を表示する行にジャンプするようになりました。

メモリ書き換えのatomicity

マルチスレッドなプログラム中でコード書き換えを行う場合には注意が必要です。書き換えを行っている最中の中途半端なコードを他のスレッドが実行してしまうという危険があります。半端な状態のコードを実行したスレッドはクラッシュ(異常終了)してしまうことでしょう。

実は上述のプログラムも、マルチスレッドプログラム中では安全とは言えません。2バイトのnopが運悪くキャッシュライン(例えば32や64バイト)をまたがってしまった場合に、書き換え中のコードが他のスレッドから見えてしまう危険が残っています。このプログラムの場合、movw命令の代わりにxchg命令を使って書き換えを行うことで安全になります。

他のスレッドから半端な状態が見えないようなひとまとまりの処理は、atomicである、と言います。どういった書き込み方がatomicとなるのかは、プロセッサの種類によってさまざまです。例えば同じx86プロセッサであっても、Pentium IIIとPentium 4では異なります。詳しくはプロセッサベンダが出している技術文書を確認する必要があります。

命令パイプラインやキャッシュの影響

近年のプロセッサは必ずと言っていいほど命令パイプラインを備えています。このため、1次(命令)キャッシュから機械語命令をフェッチ(読み込み)した後、その命令が実行されるまで、何クロックサイクルかの間があります。メモリ上の命令を書き換えても、プロセッサがその命令をすでにフェッチしてしまっている場合、実行されるのは書き換え前の命令となることがあります。このあたりの挙動はプロセッサによって異なります。

また、メモリ上の命令を書き換えた場合、データキャッシュ上の値は書き換わっても、命令キャッシュにすでに載っている値は書き換わらないというプロセッサもあります。この場合も、書き換え前の命令が実行されてしまいます。

まとめ

現代的な UNIX 系 OS の上で自分自身を書き換えるプログラムを示し、そのために必要なテクニックや注意事項を紹介しました。

—— Kazuyuki Shudo

HACK #81 SIGSEGVを使ってアドレスの有効性を確認する

“プログラマの敵”である SIGSEGV を積極的に活用して、アドレスの有効性をチェックします。シグナルハンドラに sigsetjmp(3)/siglongjmp(3)を組み合わせて使うことで、このチェックが可能となります。

本 Hack は、プログラマの敵である SIGSEGV を逆に有効利用してやろうというものです。あるアドレスが指すメモリ領域が有効であるか否かを確認します。

SIGSEGV をシグナルハンドラで捕える

アクセスする権限のないメモリ領域に対して不正にアクセスしようとすると、SIGSEGVというシグナルが発生します。通常、SIGSEGVの発生はプログラムに問題があることを意味し、これを受け取ったプログラムは "Segmentation fault" などと表示して異常終了します。

ここでは逆にSIGSEGVを積極的に活用して、アドレスの有効性をチェックします。シグナルハンドラに sigsetjmp(3)/siglongjmp(3)を組み合わせて使うことで、このチェックが可能となります。

次のプログラム中の関数 validate(void *addr)は、引数として与えられたアドレスが有効であるか否かを調べて、有効なら真、有効でなければ偽を返します。

```
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>
```

```
#define TRUE 1
#define FALSE 0

static struct sigaction orig_act;
static sigjmp_buf env;

static void sigsegv_handler(int sig) {
  siglongjmp(env, 1);
}

int validate(void *addr) {
  int is_valid = FALSE;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  struct sigaction act;

  act.sa_handler = sigsegv_handler;
  sigaction(SIGSEGV, &act, &orig_act);

  if (sigsetjmp(env, TRUE) == 0) {
    /* touch */
    volatile char c;
    c = *((char*)addr);    /* read */
    *((char *)addr) = c;   /* write */

    is_valid = TRUE;
  }
  else {
    is_valid = FALSE;
  }

  sigaction(SIGSEGV, &orig_act, NULL);

  return is_valid;
}

int main(int argc, char **argv) {
  int a;

  printf("variable a: %s\n", (validate(&a) ? "valid" : "invalid"));
  printf("100       : %s\n", (validate((void *)100) ? "valid" : "invalid"));
}
```

実行すると、次の2行が表示されます。これは、関数main()中のローカル変数aのアドレスが有効であること、論理アドレス100番地が有効ではないという実行結果を示しています。

```
variable a: valid
100       : invalid
```

説明

基本的な考え方は、試しにアクセスしてみてSIGSEGVが発生するか否かを調べる、というものです。SIGSEGVが発生した場合にもプログラムが異常終了してしまわないように、SIGSEGVをシグナルハンドラで捕らえてしまいます。SIGSEGVを捕らえておいてシグナルハンドラからただ戻ったのでは、プログラムは無限ループに陥ってしまいます。そこで、siglongjmp(3)を使って制御を戻します。

以下、関数validate()の処理を見て行きます。まず、sigaction(2)で、シグナルSIGSEGVが発生した場合に呼び出されるシグナルハンドラとして関数sigsegv_handler()を設定します。

続いてsigsetjmp(3)を呼び、実行中の状態、すなわち文脈を保存します。保存した直後はsigsetjmp(3)は0を返すので、続いて実行されるのは引数として与えられたアドレスaddrへのアクセスです。addrが指す先のメモリ領域を読み書きします。ここでaddrが有効なアドレスであれば、何事もなくアクセスは成功し、validate()は真を返すことになります。

もしaddrが有効なアドレスではなかった場合、アクセスによってSIGSEGVが発生します。すると、SIGSEGVに対するシグナルハンドラとして設定したsigsegv_handler()に制御が移ります。

sigsegv_handler()はsiglongjmp(env, 1)を実行します。これによって、先ほどsigsetjmp(3)で保存しておいた文脈、すなわちsigsetjmp(3)を呼び出したところに戻ります。この際、siglongjmp(3)の第2引数である1がsigsetjmp(3)の返り値として返されるので、関数validate()は偽を返すことになります。

こうして晴れて、有効なアドレスに対しては真を、そうではないアドレスに対しては偽を返すことができました。

前掲したプログラム中のvalidate()関数は、文脈などをstatic変数に保存するため、スレッドセーフではありません。注意してください。

まとめ

SIGSEGVを使ってアドレスの有効性を確認する手法を紹介しました。

—— Kazuyuki Shudo

HACK #82 straceでシステムコールをトレースする

straceを用いるとシステムコールをトレースすることが可能になり、プログラムのデバッグと動作の把握に大変役立ちます。

strace(http://sf.net/projects/strace/)はシステムコールをトレースするUnix用のツールです。元々はSunOS用に1991年ごろに開発が始まりました。SunOSのほか、FreeBSDやLinuxでも使うことができます。

strace の使い方

strace は Debian GNU/Linux の場合は sudo apt-get install strace でインストールできます。

straceの使い方は簡単です。基本的にはstraceコマンドの引数にトレースしたいコマンドとその引数を並べればOKです。デフォルトではstraceのメッセージは標準エラーに出力されます。これをファイルに出力させるには-oオプションを用います。たとえば、次のように実行します。

```
% strace -o log.txt emacs
```

この例ではemacsの動作中のシステムコールの呼び出しをトレースしています。emacs終了後にlog.txtを見ると、emacsがどのようなシステムコールを呼んだのかがわかります。たとえば、emacsが開いたファイルのうち、ファイル名に/home/satoruが含まれるものを一覧するには次のように実行します。

```
% grep open log.txt  | grep /home/satoru
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
  :
```

プログラムが開いたファイルを調べるのはstraceの典型的な使い方の1つです。「設定ファイルを作ったのになぜか反映されない。果たして読み込まれているのだろうか」というときはstraceを使って、その設定ファイルがプログラムによって開かれているか確認すると便利です。

strace の仕組み

strace は OS の提供するデバッグ用インタフェースを用いてシステムコールのトレースを行います。Linux では ptrace システムコールが用いられています。strace がやっていることは、基本的にはptrace(PTRACE_SYSCALL, ...)でシステムコールの入り口と出口をフックして、システムコールの引数と返り値を出力することです。

straceはシステムコールの引数と返り値を人間が読める形で表示するために多大な労力を払っています。たとえば上の例ではopenの呼び出しに対して、下のように、引数としてファイル名とフラグが表示されています。

```
open("/home/satoru/.emacs", O_RDONLY|O_LARGEFILE) = 3
```

straceではこれを実現するために、主要なシステムコールに対して専用の表示ルーチンを

用意しています。openの場合は、第1引数がパス名であること、第2引数がintの整数で表されるフラグであり、O_RDWR、O_LARGEFILEなどの定数の論理和によって渡されるという仕様に基づいて表示ルーチンが実装されています。単純にフラグを整数で表示しただけではO_RDONLY|O_LARGEFILEは32768のようになってしまいます。

まとめ

straceを用いるとシステムコールをトレースできます。プログラムのデバッグと動作の把握に大変役立つツールです。

おまけ

straceがptraceシステムコールを用いていることはstraceをstraceするとわかります。

```
% strace -o log strace -o log2 date
% grep ptrace log |head -5
ptrace(PTRACE_SYSCALL, 8744, 0x1, SIG_0) = 0
ptrace(PTRACE_PEEKUSER, 8744, 4*ORIG_EAX, [0x7a]) = 0
ptrace(PTRACE_PEEKUSER, 8744, 4*EAX, [0xffffffda]) = 0
ptrace(PTRACE_PEEKUSER, 8744, 4*EBX, [0xbfffe45c]) = 0
ptrace(PTRACE_SYSCALL, 8744, 0x1, SIG_0) = 0
```

—— Satoru Takabayashi

HACK #83 ltraceで共有ライブラリの関数呼び出しをトレースする

ltraceを使うと、共有ライブラリの関数呼び出しをトレースすることが可能になり、プログラムのデバッグと動作の把握に役立ちます。

ltraceは共有ライブラリの関数呼び出しをトレースするLinux用のツールです。システムコールをトレースするstraceと同様に、デバッグに大変役立ちます。straceについては「**[Hack #82] straceでシステムコールをトレースする**」を参照してください。

ltraceの使い方

ltraceはDebian GNU/Linuxの場合はsudo apt-get install ltraceでインストールできます。

ltraceの使い方は簡単です。基本的にはltraceコマンドの引数にトレースしたいコマンドとその引数を並べればOKです。デフォルトではltraceのメッセージは標準エラーに出力されます。これをファイルに出力させるには-oオプションを用います。たとえば、次のように

実行します。

```
% ltrace -o log.txt wget https://www.codeblog.org/
```

この例ではwgetが、https://www.codeblog.org/のコンテンツを取得するときの共有ライブラリの呼び出しをトレースしています。wgetはhttpsの通信にOpenSSLのlibssl.soを用いています。ltraceが出力したlog.txtをSSLでgrepするとどのような関数が呼び出されたかがわかります。

```
% grep SSL log.txt | head
SSL_library_init(0, 0, 0, 0, 0)                  = 1
SSL_load_error_strings(0, 0, 0, 0, 0)            = 0
OPENSSL_add_all_algorithms_noconf(0, 0, 0, 0, 0) = 1
SSL_library_init(0, 0, 0, 0, 0)                  = 1
SSLv23_client_method(0, 0, 0, 0, 0)              = 0x40038880
SSL_CTX_new(0x40038880, 0, 0, 0, 0)              = 0x808b228
SSL_CTX_set_verify(0x808b228, 0, 0x8068585, 0, 0) = 0x8068585
SSL_new(0x808b228, 0x7a060ed3, 1, 0, 0)          = 0x808cd20
SSL_set_fd(0x808cd20, 3, 1, 0, 0)                = 1
SSL_set_connect_state(0x808cd20, 3, 1, 0, 0)     = 0
SSL_connect(0x808cd20, 3, 1, 0, 0
```

「-p」オプションを用いると既存のプロセスにアタッチすることもできます。

ltraceの仕組み

ltraceの仕組みはデバッガと似ています。ltraceはデバッガと同様に、トレースの対象となるプロセスにソフトウェア的なブレークポイントを埋め込みます。上のようにltrace wget ...と実行した場合、ltraceは次のような処理を行います。

1. 環境変数PATHをたどってwgetのバイナリの絶対パスを調べる(筆者の環境では/usr/bin/wget)。
2. /usr/bin/wgetのバイナリと依存しているすべての共有ライブラリをelfutilsを用いて読み込み、関数のシンボル名とそのPLT内のアドレスのリストを取得する。
3. forkして子プロセス内でptrace(PTRACE_TRACEME, ...)をセットし、それからwgetをexecする。
4. wait()で待っている親プロセスにSIGTRAPが伝わる。
5. 親プロセスでは先ほど作っておいたリストを元に、各関数のPLTの該当アドレスにブレークポイント命令(x86では0xcc)を書き込む。このとき、書き換える前の値を保存しておく。
6. これにより子プロセスが共有ライブラリの関数を呼び出すたびにSIGTRAPが発生する

ので、親プロセスはループ内で wait で SIGTRAP を待って適宜ブレークポイントでトレースを出力しつつ、子プロセスが終了するまでループを回す。

親プロセスはブレークポイントでトレースを出力した後に、ブレークポイント命令を書き込んだアドレスの値を元に戻し、ptrace (PTRACE_SINGLESTEP, ...)で子プロセスを1命令分だけ進めます。1命令の実行が終わると、ふたたび親プロセスに制御が戻り、ブレークポイントを復元します。

PLT(Procedure Linkage Table)にはELFの共有ライブラリの関数を呼び出すときに必ず経由する非常に短いコードが各関数ごとに用意されています。ltraceはこのPLTにブレークポイントを書き込むことによって、共有ライブラリの関数呼び出しをフックしています。

まとめ

ltraceを使うと、共有ライブラリの関数呼び出しをトレースできます。プログラムのデバッグと動作の把握に大変役立つツールです。

—— Satoru Takabayashi

HACK #84 Jockey で Linux のプログラムの実行を記録、再生する

Jockey を使うと任意の Linux のプログラムの実行を記録、再生できます。Jockey はたとえば再現性の低いバグをデバッグするときに使えます。

JockeyはLinuxのプログラムの実行を記録、再生するツールです。システムコールと一部のCPU命令をフックして実行時の入出力をログに記録することによってプログラムの再生を実現しています。主にデバッグ用途に使います。

インストール

JockeyはDebianパッケージに含まれていないため、ソースコードからビルド、インストールしました。事前に ruby、boost、zlib などをインストールしておく必要があります。

使い方

Jockeyを使ってプログラムの実行を記録、再生する方法にはいくつかありますが、もっとも簡単なのは jockey コマンドを使う方法です。たとえば /bin/date の実行を記録するには次のように実行します。

```
% jockey /bin/date
Warning: /bin/date is, by default, excluded from tracing.
Warning: I'm adding 'excludedprogram=-' option as a courtesy.
2006年 1月 23日 月曜日 01:23:39 JST
```

警告が気になる場合は--excludedprogram=-をjockeyの引数に渡せば消えます。上で記録した/bin/dateの実行を再生するには次のように実行します。

```
% jockey --replay=1 /bin/date
2006年 1月 23日 月曜日 01:23:39 JST
```

何度再生を行っても、先ほどとまったく同じ時刻が表示されるところがポイントです。これは/bin/date実行を記録する際にシステムコールgettimeofday(2)の値がログに書き出され、再生時にはログに記録された値が用いられるためです。ログには他にもたくさんの情報が記録されています。

Jockeyを使用すればソケットを使ったプログラムの記録、再生もできます。

```
% jockey  wget -qO- 'http://www.random.org/cgi-bin/randnum?num=5'
17      81      81      38      18
% jockey --replay=1 wget -qO- 'http://www.random.org/cgi-bin/randnum?num=5'
17      81      81      38      18
```

この例ではrandom.orgのrandom.orgのサービスを使ってランダムな整数を5つ取得しています。再生時には記録時とまったく同じ結果が得られています。再生時には実際のネットワークアクセスを行わず、ログを元にデータの入力を再現しています。ネットワークアクセスが起きないため、再生は高速です。

仕組み

Jockeyの本体はlibjockey.soという共有ライブラリです。libjockey.soをLD_PRELOADに指定するだけで任意のプログラムを再生、記録できるため、ソースコードの編集や再リンクなどは不要です。上の実験で用いたjockeyコマンドは環境変数LD_PRELAODとJOCKEYRCを適切に設定するだけのラッパーです。

Jockeyはシステムコールと、非決定的な動きをする一部のCPU命令(現在はrdtscのみ)の呼び出す部分のコードを実行時に書き換えて(パッチを当てて)、元のシステムコール、CPU命令ではなく、ログの再生、記録が可能なJockeyが用意したコードを呼ぶようにします。これを実現するためにx86命令のテーブルベースのパーサとlibdisasm(http://bastard.sourceforge.net/libdisasm.html)というディスアセンブラライブラリを用いています。パッチを当てない部分は元のコードがそのまま実行されます。

詳しい仕組みはJockeyの論文(http://www.ysaito.com/f10-saito.pdf)で解説されています。

mmap(2)やシグナルを処理する手法など、非常に興味深い内容です。

まとめ

Jockeyを使うと任意のLinuxのプログラムの実行を記録、再生できます。Jockeyはたとえば再現性の低いバグをデバッグするときに使えます。10回に1回しか再現しないようなバグでも、Jockeyを用いて問題が再現するまで記録すれば、問題が起きたときのログを用いて何度でも再現できるようになります。再生はGDB上で行うこともできます。

このようなデバッグを行う場面はそれほど多くはないかもしれませんが、覚えておくといざというときに役立つかもしれません。役に立たないとしても技術的に面白いので、覚えておいて損はなさそうです。

—— Satoru Takabayashi

HACK #85 prelinkでプログラムの起動を高速化する

prelinkを使うと大量の共有ライブラリをリンクしたプログラムの起動時間を短縮できます。

prelinkは大量の共有ライブラリをリンクしたプログラムの起動時間を短縮するためのツールです。最近のLinuxで利用できます。「Gentoo Linux Prelinkガイド」(http://www.gentoo.org/doc/ja/prelink-howto.xml)によると「典型的なKDEプログラムの起動時間は50%も短縮することができます」とのことです。本Hackではprelinkの基本的な使い方を紹介した後に、簡単なプログラムを書いてprelinkの効果を調べる実験を行います。

prelinkとは

通常、動的リンクされたプログラムは起動時に再配置およびシンボルのルックアップを行う必要があります。prelinkは実行ファイルと依存する共有ライブラリを書き換えて、起動時にこれらの処理を行う必要性をできるかぎり減らします。また、prelinkされたプログラムはリロケーションによって発生する共有不可能なページが減るため、プロセス間で共有できるページが増えるというという効果もあります。

prelinkを使う

Debian GNU/Linuxではprelinkのパッケージをインストールすれば、cronで夜中に自動的にprelinkが実行されるように設定されます。ただし、デフォルトではprelinkは行われないので/etc/default/prelinkのPRELINKING=unknownをPRELINKING=yesに修正する必要があります。

システム全体に対してprelinkを手動で適用する場合は次のように実行します。

```
% /usr/sbin/prelink -amR
```

このコマンドを実行すると、/etc/prelink.conf に設定されているディレクトリ(/usr/bin、/usr/lib など)に含まれるすべての実行ファイルと共有ライブラリに対して prelink の処理が行われます。-a はすべて、-m は同時に使われていないライブラリのアドレスの重複を許す、R はアドレスのランダム化を行ってセキュリティを向上させるという意味です。

システム全体ではなく、特定のバイナリに対してprelinkを適用するには次のように実行します。

```
% prelink -N a.out liba.so libb.so
```

この例では実行ファイル a.out と、a.out が依存する共有ライブラリ liba.so と lib.so に対して prelink を適用します。-N は /etc/prelink.cache を更新しない、という意味です。

prelink の効果を測定する

prelink の効果を測定するために次のような実験を行いました。

1. 1,000 個の .c ファイルを生成する。個々のファイルには int funcOOXXX() { return 0; }という関数を含める。
2. 上記の .c ファイルを gcc -shared でコンパイルして 1,000 個の .so ファイルを作る。
3. 空っぽの main() しかないプログラムに上記の 1,000 個の .so ファイルをリンクする。
4. 出来上がったバイナリの実行時間を計測する。
5. prelink 後のバイナリの実行時間を計測する。
6. 静的リンクで作ったバイナリの実行時間を計測する。

結果は、次のようになりました。

prelink 前	prelink 後	静的リンク
約 4.5 秒	約 0.5 秒	約 0.0 秒

実験はコマンドラインからは次のように行いました。

```
# Debian に prelink をインストール
% sudo apt-get install prelink

# ファイルの生成とコンパイル
% time ruby prelink-test.rb
```

```
# prelink 前
% repeat 3 time ./test-dynamic
4.37s total : 4.22s user 0.15s system 100% cpu
4.62s total : 4.41s user 0.19s system 99% cpu
4.45s total : 4.28s user 0.17s system 99% cpu

# prelink を適用
% time prelink -N ./test-dynamic *.so
7.06s total : 5.52s user 1.85s system 104% cpu

# prelink 後
% repeat 3 time ./test-dynamic
0.51s total : 0.47s user 0.04s system 100% cpu
0.46s total : 0.43s user 0.03s system 99% cpu
0.49s total : 0.46s user 0.03s system 99% cpu

# 静的リンク
% time ./test-static
0.00s total : 0.00s user 0.00s system 0% cpu
```

prelink-test.rb のコードは以下の通りです。

```
system("rm -f *.c *.so")
File.open("test.c", "w") {|f|
  f.puts('int main() { return 0; }')
}

objs = []
dsos = []
1000.times {|i|
  c_file_name = sprintf("%05d.c", i)
  puts c_file_name
  File.open(c_file_name, "w") {|f|
    # f.printf('const char *s%05d = "%s";' + "\n", i, "o" * (1<<20));
    f.printf("int func%05d() { return 0; }\n", i);
  }
  obj_file_name = sprintf("%05d.o", i)
  dso_file_name = sprintf("%05d.so", i)
  system(sprintf("gcc -c %s", c_file_name))
  system(sprintf("gcc -fPIC -shared -o %s %s", dso_file_name, c_file_name))
  objs.push(obj_file_name)
  dsos.push("./" + dso_file_name)
}

system(sprintf("gcc -o test-static test.c %s", objs.join(" ")))
system(sprintf("gcc -o test-dynamic test.c %s", dsos.join(" ")))

# prelink -N test-dynamic *.so
```

まとめ

prelinkを使うと大量の共有ライブラリをリンクしたプログラムの起動時間を短縮できることがわかりました。OpenOffice.orgやFirefoxなどの大規模なアプリケーションではprelinkの効果は大きいと思います。とりわけ、C++ではシンボル名が長くなり、共通する長いprefixを持つ傾向があるため、シンボルのルックアップが減る効果は大きそうです。

ところで、筆者の環境(i386上のDebian GNU/Linux sarge)ではprelinkは0x41000000～0x50000000という240MB分の仮想アドレス空間に共有ライブラリをレイアウトします。このため、240MBに収まりきらない量の共有ライブラリをprelinkしようとすると、「Could not find virtual address slot for ./foo.so」のようなエラーになります。

参考文献

- 「prelinkの論文」(ftp://people.redhat.com/jakub/prelink/prelink.pdf)
- 「GNU C Library Version 2.3」(http://people.redhat.com/drepper/lt2002talk.pdf)
  prelinkが導入された経緯の説明があります
- 「How to Write Shared Libraries」(http://people.redhat.com/drepper/dsohowto.pdf)
  動的リンクの性能についての詳しい解説があります

—— Satoru Takabayashi

HACK #86 livepatchで実行中のプロセスにパッチをあてる

動作中のプロセスにバイナリパッチをあてるためのプログラム、livepatchについて解説します。

本Hackでは実行中のプロセスにパッチをあてるlivepatchの使い方とその原理について説明します。

livepatchとは

livepatch(http://ukai.jp/Software/livepatch/)は、実行中のユーザプロセスのコードやデータを変更することで、実行中のユーザプロセスを止めずにパッチをあてるプログラムです。

livepatchは、ターゲットプロセスのプロセスIDとターゲットプロセスのstripされていない実行バイナリを引数にとり、次のようなパッチ命令を標準入力から読みとってパッチをあてていきます。

- set アドレス タイプ値
  ターゲットプロセスのアドレスで示されるメモリに、タイプで解釈した値を書き込み

ます。

- new メモリ名 サイズ
  ターゲットプロセスにサイズバイト分のメモリを確保して、それにメモリ名という名前を付けます。

- load メモリ名 ファイル名
  ファイル名で示されるファイルの内容をターゲットプロセスにロードして、そのメモリにメモリ名という名前を付けます。

- dl メモリ名 ファイル名
  ファイル名で示される共有オブジェクトをターゲットプロセスにロードして、そのメモリにメモリ名という名前を付けます。共有オブジェクトのシンボル情報を解釈し、ターゲットプロセスと動的にリンクします。

- jmp アドレス1 アドレス2
  ターゲットプロセスのアドレス1に、アドレス2へジャンプする命令を書きこみます。

アドレスは次のようにして指定します。

- 整数値
  ターゲットプロセスにおける仮想アドレスを指します。

- $メモリ名
  new、load、dlで指定したメモリ名を使って、それらの命令で確保したターゲットプロセスのメモリ領域を指します。

- $ メモリ名:シンボル
  dlで指定したメモリ名を使って、その共有オブジェクトにおけるターゲットプロセスでのシンボルの値になります。

- $ メモリ名:整数値
  new、load、dl で確保したメモリ領域の整数値オフセットのアドレスを指します。

- シンボル名
  ターゲットプロセスのシンボルの値になります。

set で使えるタイプとしては次のようなものがあります。

- int
  値は整数値です。(strtol(value, NULL, 0)で解釈されます)
- str
  値は改行までの文字列です
- hex
  値は16進数値です。
- addr
  値はアドレスです。

livepatchの例

では簡単な例を紹介します。

まずターゲットプロセスは次のような単純な無限ループのプログラムとします。

```
% cat -n target.c
     1  #include <stdio.h>
     2  #include <sys/types.h>
     3  #include <unistd.h>
     4
     5  char *
     6  foo(int i)
     7  {
     8          static char buf[16];
     9          sprintf(buf, "%d", i);
    10          return buf;
    11  }
    12
    13  int n;
    14  char fmt[] = "foo->%s\n";
    15
    16  int
    17  main(int argc, char *argv[])
    18  {
    19          n = atoi(argv[1]);
    20          printf("pid=%d\n", getpid());
    21          while (1) {
    22                  int i;
    23                  for (i = 0; i < n; i++) {
    24                          printf(fmt, foo(i));
    25                          sleep(1);
    26                  }
    27          }
    28  }
% cc -o target target.c
```

これを実行すると、引数の数まで数字を増やしてそれを 1 行ずつ表示します。

```
% ./target 3
pid=5059
foo->0
foo->1
foo->2
foo->0
foo->1
```

このターゲットプロセスのプロセス ID は 5059 です。これに対して次のように livepatch を実行します。

```
% echo 'set n int 5' | ./livepatch  5059 ./target
```

これは、プロセス ID 5059 の target プログラムのプロセスに対して、シンボル n で指定されたアドレスに数値 5 を書き込むという意味です。これにより実行中のプロセス ID 5059 のプロセスの出力は次のように変化します。

```
foo->0
foo->1
foo->2
foo->0
foo->1      # <- livepatch でパッチをあてて n を 5 にした
foo->2
foo->3
foo->4
foo->0
foo->1
```

このように n が 3 だったものが、livepatch により 5 に変わりました。
次に foo を変更したくなったとします。

```
% cat -n foo2.c
     1  #include <stdio.h>
     2
     3  extern int n;
     4
     5  char *
     6  foo(int i)
     7  {
     8          static char buf[16];
     9          sprintf(buf, "0x0%x", n - i);
    10          return buf;
    11  }
%
```

これは次のようにして共有オブジェクトにします。

```
% cc -shared -fPIC -o foo2.so foo2.c
```

そして次のようにlivepatchコマンドを実行します。

```
% echo 'dl foo2 ./foo2.so
jmp foo $foo2:foo' | ./livepatch 5059 ./target
dl foo2 @ 0xb7fe6000 [6368] ./foo2.so
jmp 0x8048454 0xb7fe6714
```

これはfoo2.soという共有オブジェクトをロードして、foo2という名前を付け、ターゲットプロセスにリンクし、ターゲットプロセスのfooにfoo2という共有オブジェクト内のfooというシンボルへジャンプする命令を書き込みます。つまり、元のターゲットプロセスのfoo()のエントリに、新しくロードしたfoo2.soの中のfoo()へのジャンプを書き込むことで、元のfoo()の呼び出しはfoo2.soのfoo()の実行になるようにするわけです。これにより実行中のプロセスID 5059は次のように出力が変わります。

```
foo->0
foo->1
foo->2
foo->0x02        # livepatch でfoo()を変更した
foo->0x01
foo->0x05
foo->0x04
foo->0x03
foo->0x02
foo->0x01
```

livepatchの原理

livepatchコマンドは特殊なカーネルを必要としません。ptrace(2)というデバッガ用のシステムコールを使って、ターゲットプロセスのメモリの読み書きを行なっています。そしてlibbfdを使ってターゲットコマンドおよび共有オブジェクトのシンボルテーブルを読みとりリンクしています。

ptrace(2)の使い方

ptrace(2)を使って、ターゲットプロセスのメモリの読み書きをするにはまずそのターゲットプロセスにアタッチする必要があります。アタッチするにはターゲットプロセスのプロセスIDに対してPTRACE_ATTACHというリクエストをptrace(2)を使って発行し、ターゲットプロセスがptraceリクエストを受け取って停止するのをwait(2)を使って待ちます。

```
680      if (ptrace(PTRACE_ATTACH, target_pid, NULL, NULL) < 0) {
681          perror("ptrace attach");
```

```
682         exit(1);
683     }
684     DEBUG("attached %d\n", target_pid);
685     wait(NULL);
```

アタッチできたら、PTRACE_PEEKDATAおよびPTRACE_POKEDATAを使ってターゲットプロセスのメモリを読み書きします。

```
448     int *lv = malloc(len * sizeof(int));

455         lv[i] = ptrace(PTRACE_PEEKDATA, pid,
456                        addr0 + i * sizeof(int), NULL);
```

この場合、pid で示されるターゲットプロセスのメモリ addr0 + i * sizeof(int)にある int 値を読みとります。

```
469         if (ptrace(PTRACE_POKEDATA, pid,
470                    addr0 + i * sizeof(int), lv[i]) < 0) {
471             perror("ptrace poke");
472             return -1;
473         }
```

これで、pid で示されるターゲットプロセスのメモリ addr0 + i * sizeof(int)に lv[i]で表される int 値を書き込みます。メモリを読み書きする以外にターゲットプロセスでのレジスタの読み書きも行うことができます。

```
362     struct user_regs_struct regs, oregs;

367     if (ptrace(PTRACE_GETREGS, pid, NULL, &oregs) < 0) {
368         perror("ptrace getregs");
369         return 0;
370     }
371
372     regs = oregs;
```

これでpidで示されるターゲットプロセスのレジスタをoregsに読みとってきます。x86アーキテクチャの場合、regs.esp で %esp が、regs.eax で %eax などが読みとれます。

書き込む時には次のようにします。

```
412     if (ptrace(PTRACE_SETREGS, pid, NULL, &regs) < 0) {
413         perror("ptrace set regs");
414         return 0;
415     }
```

ターゲットプロセスにコードを実行させることもできます。

```
417     if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0) {
418         perror("ptrace cont");
419         return 0;
420     }
421     wait(NULL);
```

PTRACE_CONTの前にPTRACE_SETREGSでregs.eipに次に実行するコードの先頭アドレスを指定しておくとそこにあるコードの実行が開始されます。そしてそのコードの中にブレークポイントとなる命令(int3、0xcc)を埋め込んでおくとその時点でターゲットプロセスは停止して、アタッチしていたプロセスがwaitで待っていたのが起こされて処理を続けます。

最後のPTRACE_DETACHでデタッチします。

```
878     ptrace(PTRACE_DETACH, target_pid, NULL, NULL);
879     DEBUG("detached %d\n", target_pid);
```

ptrace(2)を使ってメモリ領域の確保

ptrace(2)だけではターゲットプロセスに新たにメモリを確保することはできません。ではlivepatchのnew、load、dl命令はどのようにしているのでしょうか?

実はここまでに述べたptraceリクエストを使って次のようなコードをターゲットプロセスに送り込んで実行させています。

```
mmap(NULL, siz, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

これは引数をスタックに設定し、%eaxにシステムコール番号SYS_mmapを設定し、int $0x80というコードを実行させれば実現できます。int $0x80のあとにint3を入れておけばint $0x80から戻ってきた時点でブレークポイントになりターゲットプロセスは停止し、livepatchプロセスのほうに制御が戻ります。この時点でmmap(2)が確保したメモリ領域の先頭アドレスがmmap(2)のリターン値として%eaxに設定されているので、それを読みとることでターゲットプロセスのどこにmmap(2)されたのかがわかるわけです。

外部動的リンク

livepatchはターゲットプロセスの外から動的リンクをしています。

まず/proc/ターゲットプロセスのPID/mapsを読みとり、共有ライブラリがどのアドレスにマップされているかを調べています。/proc/$$/mapsは次のようなフォーマットになっています。

```
% cat /proc/5059/maps
08048000-08049000 r-xp 00000000 03:01 5081628    /tmp/target
08049000-0804a000 rw-p 00000000 03:01 5081628    /tmp/target
```

```
b7ea3000-b7ea4000 rw-p b7ea3000 00:00 0
b7ea4000-b7fce000 r-xp 00000000 03:01 36834      /lib/tls/i686/cmov/libc-2.3.2.so
b7fce000-b7fd7000 rw-p 00129000 03:01 36834      /lib/tls/i686/cmov/libc-2.3.2.so
b7fd7000-b7fd9000 rw-p b7fd7000 00:00 0
b7fe8000-b7fea000 rw-p b7fe8000 00:00 0
b7fea000-b8000000 r-xp 00000000 03:01 2277452    /lib/ld-2.3.2.so
b8000000-b8001000 rw-p 00015000 03:01 2277452    /lib/ld-2.3.2.so
bfffe000-c0000000 rw-p bfffe000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

それぞれの行は次のような情報を含んでいます。

```
vm_start vm_end  flags pgoff  major:minor ino    ファイル名
```

そしてターゲットプログラムおよびそれが利用している共有ライブラリを、libbfdを使って読みとって定義されているシンボルとその値をbfd_canonicalize_symtab()とbfd_canonicalize_dynamic_symtab()で読みとります。そしてマップされているアドレスに従ってターゲットプロセスでのメモリアドレスに変換しています。

dl命令で、共有オブジェクトをロードする時に、その共有オブジェクトの未定義シンボルは、ターゲットプロセスのシンボルの値とバインドして動的リンクする必要があります。ターゲットプロセスのシンボルの値は前記のように得ることができているので、あとは共有オブジェクトのリロケーションテーブル(.rel.dyn と .rel.plt)を書き変えています。

まとめ

動作中のプロセスのメモリやレジスタは、デバッグ用のシステムコールptrace(2)を使うことで変更することができます。それを利用して開発された、動作中のプロセスにバイナリパッチをあてるためのプログラム、livepatch について説明しました。

—— Fumitoshi Ukai

6章

プロファイラ・デバッガ Hack

Hack #87-92

本章ではプロファイラとデバッガに関するテクニックを紹介します。コンピュータの性能はCPUやメモリの性能の向上に支えられて飛躍的に向上してきました。しかしながら、膨大なトラフィックをさばくウェブサイトの運営や、データベース、動画処理といった場面では、いくら性能があっても充分ではありません。高速なプログラムを書くことはプログラマにとってまだまだ重要な課題の1つです。また、バグのないプログラムを書くことはプログラマの最重要課題であることは言うまでもありません。

本章では、まず高速なプログラムを開発する上で役立つ各種のプロファイラの基本的な使い方と仕組みについて紹介します。次に、デバッガを使う上で役に立つテクニックを紹介します。

HACK #87 gprof でプロファイルを調べる

プロファイリングとは、プログラムのどの部分で処理に時間がかかっているか調べることです。本 Hack では、gprof でプロファイルを調べる方法を紹介します。

通常のプログラムでは、パフォーマンス上のボトルネックとなっている部分はごく一部のコードであることが多いものです。プログラムの遅い部分を割り出すためにプログラムのどの部分で時間がかかっているか調べることをプロファイリングと言います。本 Hack では、gprof でプロファイルを調べる方法を紹介します。

使い方

以下のような簡単なプログラムのプロファイルを調べてみます。

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}

void f() {
```

```
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
}

int main() {
    f();
    g();
}
```

以下のようにコンパイルしてgprofを使用します。

```
% gcc -O -g -pg bench.c
% ./a.out
% gprof a.out
```

gccに-pgオプションを付けてプロファイル情報を出力するようにしておくことに注意して下さい。./a.outを実行すると、gmon.outというファイルにプロファイル情報が書き出されます。gprof a.outの出力の一部を以下に示します。

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     25.86    25.86     5000     0.01     0.01  slow
  0.00     25.86     0.00        1     0.00     5.17  f
  0.00     25.86     0.00        1     0.00    20.69  g

index % time    self  children    called           name
                5.17    0.00    1000/5000          f [4]
               20.69    0.00    4000/5000          g [3]
[1]    100.0   25.86    0.00    5000             slow [1]
-----------------------------------------------
                                                <spontaneous>
[2]    100.0    0.00   25.86                    main [2]
                0.00   20.69       1/1             g [3]
                0.00    5.17       1/1             f [4]
-----------------------------------------------
                0.00   20.69       1/1          main [2]
[3]     80.0    0.00   20.69       1            g    [3]
               20.69    0.00    4000/5000       slow [1]
-----------------------------------------------
                0.00    5.17       1/1          main [2]
[4]     20.0    0.00    5.17       1            f    [4]
                5.17    0.00    1000/5000       slow [1]
-----------------------------------------------
```

最初のテーブルでは関数ごとの所要時間が示されます。slow関数がほとんどの時間を消費している（self seconds）こと、gはfの4倍の回数slowを呼び出しているため、トータル消費時間（total s/call）もほぼ4倍になっています。

2番目のテーブル（コールグラフ）では、関数の呼び出し関係を調べることができます。mainからfとgが呼ばれ、fとgがslowを呼んでいるということが見てとれます。

gprofに-lを付けると行ごとの所要時間を調べることができます。これには-gオプションでデバッグ情報を付けることが必須です。-l -A -xと付けるとソースコードを表示の上、行ごとの所要時間を見ることができます。特定の関数のコールグラフのみを見たい場合は、gprof -F slow などと指定します。

gcc -pgを用いると、各処理ごとにカウンタを増やす処理が埋め込まれることに注意して下さい。これによって通常実行時よりもパフォーマンスが落ちることになります。

原理

gcc-pg で出力されるバイナリは3種類のプロファイル情報を残します。デフォルトではgmon.outというファイルに残されますが、GMON_OUT_PREFIX環境変数によって、このファイル名を変更することもできます。このファイルの情報は「GNU gprof - Implementation」（http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_node/gprof_25.html）で詳しく説明されています。

まず、1つ目はタイマによって調べられている10ミリ秒ごとでのプログラミングカウンタの位置です。これは「**[Hack #31]　main() の前に関数を呼ぶ**」で紹介した方法でsetitimer(2)をmainの前に呼び出し、10ミリ秒ごとにSIGPROFが発生するようにしています。SIGPROFで呼ばれたハンドラ（glibcのsysdeps/posix/sprofil.cのprofil_count）ではカウンタを加算しています。実際にはglibcのprofil(3)を用いることによってこれは実現されています。

2つ目は関数のコールグラフ情報です。これは「**[Hack #77]　関数へのenter/exitをフックする**」で紹介したような方法で、関数に入る時にmcount 関数が呼ばれるようにしています。mcount ではアーキテクチャ依存な短いアセンブラ（glibcのsysdeps/i386/i386-mcount.Sなど）で、この中で呼び出し元PCと呼び出し後のPC情報を取得した後にアーキテクチャ非依存な__mcount_internalを呼び出します。そしてPCの情報を用いてコールグラフ情報を作成しつつ、正確な関数呼び出し回数を記録します。

3つ目は古いGCCを使用した時に必要になる情報で、基本ブロック（ifブロックやwhileブロックなど）ごとにそのブロックに来た回数を逐一記憶します。このカウンタはコンパイラによって作成されます。このカウンタによって完全なカバレッジ情報が記録されますが、動作が非常に遅くなるため、時間的なプロファイル情報はかなり不正確になるでしょう。現在はこの情報は記録されていません。カバレッジを知りたい時にはgcovを使用するとよいでしょう。

gprof コマンドを実行すると、これらの情報が記録されているファイルを開いて、正確なコールグラフと関数呼び出し回数情報を取得し、また、サンプリング結果を元に算出した大ざっぱな所要時間を得ることができます。さらに、「**[Hack #67] libbfdでシンボルの一覧を取得する**」で紹介したような方法で実行ファイルからアドレス→シンボル名の情報を取得し、それらをまとめて表示します。

まとめ

本 Hack では、gprof でプロファイルを取る方法と、その原理を紹介しました。

—— Shinichiro Hamaji

HACK #88 sysprofでお手軽にシステムプロファイルを調べる

sysprof を使えば、非常に手軽に簡単なプロファイルを取ることができます。

本Hackではシステム全体のプロファイルを取るソフトウェアである、sysprofを紹介します。

インストール

執筆時点でのsysprof（http://www.daimi.au.dk/~sandmann/sysprof/）のバージョンは1.0.2です。ライセンスは GPL2 となっています。

sysprofは、Linuxカーネルモジュールと GUIのクライアントアプリケーションの組み合わせで動作するアプリケーションです。現状では x86 と x86-64 だけがサポートされています。

まずはLinux カーネルのプロファイリングサポートをオンにする必要があります。使用中のカーネルがこの機能をサポートしていない場合は、カーネル2.6.11以降で、CONFIG_PROFILING の設定をオンにして再コンパイルして下さい。ついでにCONFIG_OPROFILEもオンにしておくと「**[Hack #89] oprofile で詳細なシステムプロファイルを得る**」でも使用できて良いでしょう。また、GUI のために GTK+ 2.6.0 以降と libglade 2.5.1 以降が必要となります。

そこまですませれば、後は sysprof のアーカイブを展開して ./configure; make; make install するだけでインストールできます。あとはmodprobe sysprof-moduleなどとしてモジュールをロードすれば、準備は完了です。

使い方

sysprofは使用法を説明する必要がないほど簡単に使用できます。特定のプログラムのプロファイルを取りたい場合、sysprof を動かして Start ボタンをクリックしたのち、そのプログ

ラムを動かして、止まったところで Profile ボタンを押せばいいでしょう。ここでは「**[Hack #87] gprof でプロファイルを調べる**」と同様の、以下のサンプルを使用してみます。

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}

void f() {
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
}

int main() {
    f();
    g();
}
```

特にコンパイルオプションは付ける必要はありませんし、デバッグシンボルもなくてかまいません。シンボルを strip してしまわなければ OK です。

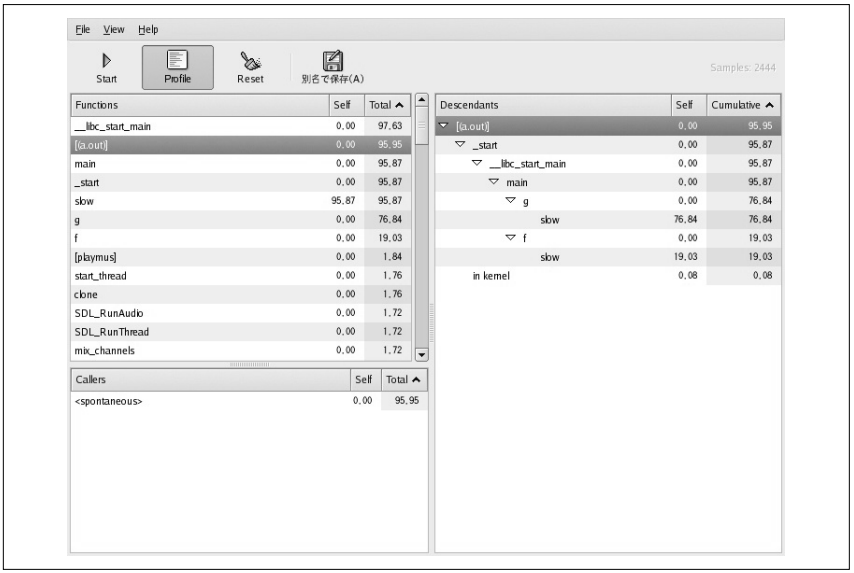


図 6-1　sysprof の実行画面

このサンプルのプロファイルを取った時のスクリーンショットが図6-1です。ここでもだいたい関数fとgが2:8くらいになっています。また、システム全体のプロファイラであるため、ちょうどこの時は音楽を聞いていたことから、playmusというソフトウェアもリストに入っていることがわかります。このプロファイルはxmlファイルで保存することもできます。

sysprofは、動作中に高頻度でサンプリングを取って、その時々のプログラムカウンターの位置を記録していく動作になっているため、必ずしも完全に正確な結果を返すわけではありませんが、プログラムのボトルネックを探すだけであれば十分に実用に耐えるでしょう。

まとめ

本Hackでは、sysprofでシステム全体のプロファイルを取る方法を紹介しました。sysprofを使えば、非常に手軽に簡単なプロファイルをとることができます。

—— Shinichiro Hamaji

HACK #89 oprofile で詳細なシステムプロファイルを得る

本 Hack ではシステムプロファイルを取るソフトウェアである、oprofile を紹介します。oprofile は多くの CPU に対応し、sysprof よりも詳しい CPU の情報を得ることが可能です。

「**[Hack #88]　sysprof でお手軽にシステムプロファイルを調べる**」で紹介した sysprof は用途が限定的なものの非常に簡単に扱えるソフトウェアでした。本Hackで紹介するoprofileは対照的に、比較的複雑な操作を必要としますが、多くの CPU で動作し、CPU の種々の機能を用いてさまざまなシステムプロファイルを取得することができます。

インストール

執筆時点でのoprofileのバージョンは0.9.1です。oprofileは、GPL2のもとで配布されています。sysprofと同様、oprofileもカーネルモジュールとクライアントアプリケーションが協調して動作するソフトウェアです。ドキュメントによると、Linux 2.2ではx86が、Linux 2.4では加えてIA-64が、Linux 2.6では加えてAlpha、MIPS、ARM、x86-64、sparc64、ppc64と限定的ですがPA-RISCとs390がサポートされているそうです。

筆者は oprofile を、Mobile Celeron 1.7GHz、Pentium 4 2.53GHz、Xeon 3.2GHz デュアルプロセッサという 3 つの環境で試しました。Xeon では豊富な CPU 情報がすべて使用でき、Pentium 4 では簡単な情報のみが取得でき、Mobile Celeron ではプロファイルを取得できませんでした。oprofileのサイトによると、ラップトップマシンでは動作しない場合があるようです。本 Hack は Xeon のマシンを用いて動作確認を行いました。

sysprof の Hack（**[Hack #88]**）で紹介した通り、カーネルの設定で `CONFIG_PROFILING` と

CONFIG_OPROFILE をオンにしておき、oprofile を通常通り ./configure; make; make install すればインストールは完了です。また、カーネル自身を oprofile で測定する場合、CONFIG_FRAME_POINTER オプションもオンにしておくとカーネルのコールグラフが取れて良いでしょう。

基本的な使用法

oprofile はプロファイルを取るまでは opcontrol で制御し、プロファイルの結果を閲覧する時は opreport や opannotate などを使用します。oprof_start という Qt を用いた GUI フロントエンドも付属していますが、筆者が試してみたところ、あまり使いやすくなかったため、本 Hack では CUI のコマンドのみを解説します。

まず、基本的な設定を済ませると良いでしょう。最低限以下のようにして、カーネルの位置（カーネルを渡す際は、圧縮されたイメージである vmlinuz ではダメなことに注意して下さい）と記録するコールグラフの数くらいは設定しておくと良いでしょう。

```
% opcontrol --vmlinux=/usr/src/linux-2.6.15.4/vmlinux --callgraph=20
```

これらの設定は $HOME/.oprofile/daemonrc に記録されますので、一度設定すると変更する必要はなくなります。以下で、デーモンプロセス oprofiled を起動することができます。

```
% opcontrol --start
```

また、プロファイルを記録する場合には以下を使用します。

```
% opcontrol --dump
```

いったんプロファイルのカウントをリセットする場合は --reset オプションを使用し、いったん止める場合は --stop、デーモンを終了する場合は --shutdown オプションを使用します。

gprof や sysprof の Hack（**[Hack #87]**、**[Hack #88]**）と同様、以下のコードのベンチマークを取ってみましょう。

```
void slow() {
    int i;
    for (i = 0; i < 2000000; i++);
}

void f() {
    int i;
    for (i = 0; i < 1000; i++) slow();
}

void g() {
    int i;
    for (i = 0; i < 4000; i++) slow();
```

```
}

int main() {
    f();
    g();
}
```

oprofiled が動いている状態で、以下のように操作します。

```
% gcc -O -g bench.c
% opcontrol --reset
% ./a.out
% opcontrol --dump
```

これでプロファイルがファイルに保存されたはずです。opreport を用いてプロファイルを見てみましょう。さまざまなオプションがサポートされていますが、ここでは-cでコールグラフを見てみます。

```
% opreport -c

samples  %        image name               app name                 symbol name
-------------------------------------------------------------------------------
  5         0.0051  a.out                    a.out                    main
  19409    19.9324  a.out                    a.out                    f
  77960    80.0624  a.out                    a.out                    g
  97374    53.1753  a.out                    a.out                    slow
  97374    100.000  a.out                    a.out                    slow [self]
-------------------------------------------------------------------------------
  12937     7.0648  libruby.so.1.8.3         libruby.so.1.8.3         (no symbols)
  12937    100.000  libruby.so.1.8.3         libruby.so.1.8.3         (no symbols) [self]
... 略 ...
```

今までと同様、f:g が 2:8 くらいの割合で時間を使っていることがわかります。CPU が提供している情報を使用しているからでしょう、sysprofと比較して非常に多くのサンプルを取得していることもわかります。また、sysprofと同様システム全体のプロファイルをとるソフトウェアですので、下には libruby.so.1.8.3 などと、別プロセスの情報も見えています。

次に opannotate を用いてソースコードレベルでのプロファイル情報を見てみましょう。opannotateのオプションに-aを付けるとアセンブリコードが出力され、-sを付けるとソースコードが出力されます。ソースコード表示にはデバッグ情報が必要となります。どちらも付けるのが一番わかりやすいでしょう。

```
% opannotate -a -s
... slow 関数のみを示します ...
08048348 <slow>: /* slow total:  97374  5.6869 */
              :void slow() {
              : 8048348:       push   %ebp
```

```
                :  8048349:       mov    %esp,%ebp
                :  804834b:       mov    $0x1e8480,%eax
                :      int i;
                :      for (i = 0; i < 2000000; i++);
 97361  5.6861 :  8048350:       dec    %eax
     7 4.1e-04 :  8048351:       jne    8048350 <slow+0x8>
                :}
     1 5.8e-05 :  8048353:       leave
     5 2.9e-04 :  8048354:       ret
```

キャッシュミスを検出する

oprofileを用いると、プロファイル情報だけでなくさまざまなイベントの起きた回数を調べることもできます。検出できるイベントのリストはoprofile --list-eventsで見ることができます。キャッシュミスを検出するためには、BSQ_CACHE_REFERENCEを調べると良いでしょう。調べるイベントを増やすには例えば以下のようにします。

```
% opcontrol --event=BSQ_CACHE_REFERENCE:100000:0x100
% opcontrol --shutdown
% opcontrol --start
```

100000という数字はサンプルレートで、0x100はユニットマスクと呼ばれる、拾うイベントを選択するためのものです。0x100を指定すると読み込み時のL2キャッシュミスを拾うはずです。この数値は、「IA-32 Intel Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 2」のAppendix A.4 Table A-11に載っています。今回調べるコードは以下のような簡単なサンプルです。

```
#include <stdlib.h>
#define NUM 10000
int cache_miss() {
    int* a[NUM];
    int i, j, sum;
    for (i = 0; i < NUM; i++) a[i] = (int*)malloc(sizeof(int)*NUM);
    for (i = 0; i < NUM; i++)
        for (j = 0; j < NUM; j++)
//            sum += a[i][j];
            sum += a[j][i];
    return sum;
}
int main() {
    return cache_miss();
}
```

これは二次元配列の中身をすべて足し合わせていくコードですが、a[j][i]でアクセスしているため、アクセスする順序がa、a+10000、a+20000...、a+1、a+10001...となるため、コメン

トアウトされている a[i][j]のバージョンよりもキャッシュミスが起こりやすいはずです。

```
% opcontrol --reset
% ./a.out
% opcontrol --dump
% opreport -c

samples  %        image name               app name                 symbol name
-------------------------------------------------------------------------------
  231     100.000  a.out                    a.out                    main
  231     94.6721  a.out                    a.out                    cache_miss
  231     100.000  a.out                    a.out                    cache_miss [self]
```

合計231回のキャッシュミスが起きたようです。次にコメントアウトする部分を交換して実行した結果を示します。

```
  1       100.000  a.out                    a.out                    main
  1       10.0000  a.out                    a.out                    cache_miss
  1       100.000  a.out                    a.out                    cache_miss [self]
```

キャッシュミスは1回と、非常に少なくなりました。予想通りの結果と言えるでしょう。

まとめ

本 Hack ではシステムプロファイルを取るソフトウェアである、oprofile を紹介しました。

—— Shinichiro Hamaji

HACK #90 GDB で実行中のプロセスを操る

本 Hack では、GDB が持つ ptrace(2)システムコールの機能を積極的に活用して、実行中のプロセスを操る例をいくつか紹介します。

GDBはptrace(2)システムコールの便利なフロントエンドインタフェースとして使えます。特にターゲットプロセス内の関数を式の中で呼ぶことができる機能は強力で、ptraceのアタッチ機能と組み合わせて使えば、実行中のプロセスに対してちょっかいを出すことが簡単にできます。

活用例

単純な活用例をいくつか紹介します。以下はx86のDebian GNU/Linuxのシステムで実験しました。なお GDB は 6.4 より前のバージョンではうまく動かない例があります。

```
% ps x | grep firefox
 3616 ?        Rl    19:40 /usr/lib/firefox/firefox-bin -a firefox
% gdb -q -p 3616
```

```
(gdb) p chdir("/")
[Switching to Thread -1221168480 (LWP 3616)]
$1 = 0
(gdb) detach
Detaching from program: /usr/lib/firefox/firefox-bin, process 3616
(gdb)
```

この例はgdb -pでアタッチしたターゲットプロセス内でchdir(2)システムコールを呼び出しています。デーモンやXクライアントといったバックグラウンドプロセスがマウントポイントの下にいるおかげで、device is busy で umount できないといった状況で有用です。

ファイル操作も比較的簡単に行うことができます。まずcatを起動しておき、別のシェルでは以下のように GDB を起動して cat プロセスをいじってみます。

```
% gdb -q -p $(pidof cat)
(gdb) p write(1, "hoge", 4)
$1 = 4
(gdb) p open("/etc/passwd", 0)   ← 0 は O_RDONLY です
$2 = 3
(gdb) p dup2(3, 0)
$3 = 0
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

catプロセスがhogeという文字列と/etc/passwdファイルの内容を出力していれば成功です。気合いと根性があればソケットを使って新たにTCPコネクションを張ることも可能でしょう。

execlp(2)を呼んで、途中から別のコマンドにしてしまうこともできます。

```
(gdb) p execlp("ls", "ls", "/", 0)

Program exited normally.
The program being debugged stopped while in a function called from GDB.
When the function (execlp) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
(gdb)
```

"ls /"が実行されたはずです。最後のメッセージは、GDBが想定した場所に制御が戻ってこなかったことを警告しています。この他にもたとえば、ターゲットプロセスがlibdlをリンクしているプログラムなら、dlopen(3)を呼んで共有オブジェクトをリンクして使うことができます。しかしlibdlがリンクされていなかった場合、dlopen(3)は使えず、GDBにもリンカの機能はないので、代わりに **[Hack #86]** の livepatch のようなリンク機能を備えたプログラムに頼ることになるでしょう。

応用例：外部コマンド cd

Unix に関してよく尋ねられる質問に、なぜ cd は外部コマンドでなくシェルのビルトインなのか、というものがあります。それは他のプロセスのカレントディレクトリを変更することはできないからだ、というのが答えなわけですが、ここでは本 Hack の応用例として、あえて外部コマンドで cd を実装してみます。

実装は GDB とスクリプト言語を組み合わせれば簡単にできます。

```
#!/usr/bin/ruby
# Usage:
#   ext-cd directory

require 'tempfile'

dir = ARGV.shift

t = Tempfile.new("ext-cd.gdb")
t.puts <<"End"
attach #{Process.ppid}
call chdir(#{dir.dump})
End
t.close

# shut up gdb.
STDIN.reopen("/dev/null")
STDOUT.reopen("/dev/null")
STDERR.reopen("/dev/null")

system("gdb", "-batch", "-n", "-x", t.path)
```

実行結果は例えば次のようになります。

```
$ /bin/pwd
/tmp
$ ./ext-cd /
$ /bin/pwd
/
```

なお余談として、Solaris には /bin/cd がありますが、このコマンドは ext-cd のような動作はしません。単に cd プロセスの中で chdir するだけです。これはすべてのコマンドは exec(2) できると POSIX で決まっているためですが、その理由は POSIX を決めるときの議論で、どんなに外部コマンドとしては役に立たなさそうなコマンドでも、誰かが役に立つケースを持ち出して議論がまとまらず、結果としてすべてのコマンドを外部コマンドとしても用意することになったという話です。

まとめ

本 Hack では、GDB が持つ ptrace(2) システムコールの機能を積極的に活用して、実行中のプロセスを操る例をいくつか紹介しました。GDB を使えば、Ruby のようなスクリプト言語からでも簡単にそのようなプログラミングができます。

—— Takeshi Yaegashi, Akira Tanaka

HACK #91 ハードウェアのデバッグ機能を使う

x86 の持つハードウェアデバッグ支援機能の使用法を解説します。

プロセッサの中にはデバッグ支援機能をハードウェアで持つものがあり、例えばx86アーキテクチャでは 8 本のデバッグレジスタ（DR0 ～ DR7）というものが用意されています。本 Hack では Linux プロセスでこれを活用する方法について説明します。

x86 のデバッグレジスタ

x86のデバッグ機能については「IA-32 Intel Architecture Software Developer's Manual, Volume 3B」の「CHAPTER 18 Debugging and Performance Monitoring」に完全な解説があります。

簡単に説明すると、DR0-DR3の4つのレジスタで指定したリニアアドレスの示すメモリ領域にプロセッサのアクセスがあると、INT 1のデバッグ例外を発生させてくれるというものです。残りの 4 つのレジスタは予約か、デバッグ支援機能の制御用となっています。

Linux/x86 ではデバッグレジスタはプロセスごとにその値を設定でき、INT 1を発生させたプロセスにはSIGTRAPが送られることになっています。これを利用してGDBはハードウェアウォッチポイントを実現しています。

自分のデバッグレジスタを書き換える

デバッグレジスタへのアクセスはカーネルモードでしか許されていないため、プロセスがデバッグレジスタを書き換えるにはptraceシステムコールを使う必要があります。カーネルヘッダファイル asm-i386/user.h を見ると struct user に int u_debugreg[8]; というメンバが定義されています。

GDBがターゲットプロセスのデバッグレジスタを読み書きする場合はこれでよいのですが、プロセスが自分自身のデバッグレジスタを読み書きしたい場合、プロセスは自分自身に対して ptrace することができません。そこで、fork を使って子プロセスに ptrace をしてもらいます。

以下はデバッグレジスタに値をセットする関数の実装例です。

```
#include <asm/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void
set_debugregs(unsigned long *v)
{
    if (!fork()) {
        int i, *p = ((struct user *)0)->u_debugreg;
        pid_t ppid = getppid();
        ptrace(PTRACE_ATTACH, ppid, NULL, NULL);
        waitpid(ppid, NULL, 0);
        for (i = 0; i < 8; i++, p++, v++) {
            if (i == 4 || i == 5)
                continue;
            if (ptrace(PTRACE_POKEUSER, ppid, p, *v) < 0)
                fprintf(stderr,
                    "ptrace failed: dr%d = %08lx\n", i, *v);
        }
        ptrace(PTRACE_DETACH, ppid, NULL, NULL);
        exit(0);
    }
    wait(NULL);
}
```

このように値をセットする関数は、比較的簡単に書くことができますが、逆に値を読み出したい場合は子プロセスが行ったptrace(PTRACE_PEEKUSER)の結果を得るためになんらかのプロセス間通信を行う必要があるでしょう。

デバッグレジスタの活用例

上のset_debugregs()関数を使った簡単なテストプログラムで実験してみましょう。

```
#include <signal.h>
#include <asm/ucontext.h>

int tmp, data0, data1;
void func(void) {}

static unsigned long regs[] = {
        (unsigned long)&data0,
        (unsigned long)&data1,
        (unsigned long)func,
```

```
        0, /* unused */
        0, /* reserved */
        0, /* reserved */
        0, /* cant read */
        0x00fd013f, /* Trap conditions
                       DR0: write, DR1: read/write, DR2: exec, DR3: unused */
};

#define TRY(x) do { fputs("Trying " #x "\n", stderr); x; } while (0)

static void
trap_handler(int n, siginfo_t *si, struct ucontext *uc)
{
        fprintf(stderr, "  Trapped at 0x%08lx\n", uc->uc_mcontext.eip);
        /* When we hit DR2, disable breakpoints to avoid infinite loop  */
        if (uc->uc_mcontext.eip == regs[2]) {
                regs[7] = 0;
                set_debugregs(regs);
        }
}

int
main(void)
{
        struct sigaction sa = {
                .sa_sigaction = (void *)trap_handler,
                .sa_flags = SA_RESTART | SA_SIGINFO,
        };

        sigemptyset(&sa.sa_mask);
        sigaction(SIGTRAP, &sa, NULL);
        set_debugregs(regs);

        TRY(tmp = data0);
        TRY(tmp = data1);
        TRY(data0 = 1);
        TRY(data1 = 1);
        TRY(func());

        return 0;
}
```

コンパイルして実行すると以下のようになります。

```
% gcc -g -Wall -O2 debugregs.c
% ./a.out
Trying tmp = data0
Trying tmp = data1
  Trapped at 0x080487de
Trying data0 = 1
  Trapped at 0x08048818
Trying data1 = 1
```

```
  Trapped at 0x08048844
Trying func()
  Trapped at 0x080486b0
```

このように、x86のデバッグレジスタを使えば、一般的なページ単位のメモリ保護機構には不可能な、微細なメモリスポットのアクセスも感知できることがわかります。

またGDB中でこのプログラムを動かせば、SIGTRAPが発生した時点でGDBに制御が移ります。「**[Hack #92] Cのプログラムの中でブレークポイントを設定する**」で紹介しているテクニックと同様に、プログラム中で動的にウォッチポイントを設定したい場合に使えます。

まとめ

x86の持つハードウェアデバッグ支援機能の使用法を説明しました。監視できるのは最大4ワードというごく少量のメモリではありますが、うまく使えばなにかの役には立つでしょう。

参考文献

- 『IA-32 Intel; Architecture Software Developer's Manual, Volume 3B』の「CHAPTER 18 Debugging and Performance Monitoring」(http://www.intel.com/design/Pentium4/manuals/253669.htm)
- 『図解32ビットマイクロコンピュータ80486の使い方』の「13章 デバッグサポート」(W. B. スルヤント著、オーム社)

—— Takeshi Yaegashi

HACK #92 Cのプログラムの中でブレークポイントを設定する

Cのプログラムをデバッグする際にブレークポイントを設定する方法を紹介します。

Cのプログラムをデバッグする際にはGDBなどのデバッガが役に立ちます。通常、ブレークポイントはデバッガの中から設定しますが、本Hackではデバッグ対象のCのプログラムの中でブレークポイントを設定する方法を紹介します。

Linuxなら#include <signal.h>して、任意の箇所に以下を挿入すればOKです。

```
raise(SIGTRAP);
```

raise()関数を用いてSIGTRAPシグナルを発生させています。あるいはx86限定なら以下でもOKです。

```
__asm__ __volatile__("int3");
```

ここではSIGTRAPを発生させるためにint3 (0xcc)命令を埋め込んでいます。この方法の場合、raise()の呼び出しと違い、関数呼び出しが発生しないため、コールスタックが乱れないというメリットがあります。GDBもソフトウェア的にブレークポイントを設定するときは当該箇所にint3を書き込んでいるので、やっていることは似ています(GDBの場合はint3を書き込む部分の元のコードを保存しておいたりする必要がありますが)。

上のコードを仕込んでコンパイルしたバイナリをGDB上で動かすと、当該箇所で処理が中断して制御がGDBに移るはずです。

デバッガ上で実行されているか調べる

プログラムがデバッガ上で実行されている場合、SIGTRAPはデバッガによって処理される、という性質を利用すると、プログラムがデバッガ上で実行されているか調べることができます。以下のプログラムはデバッガ上で実行されている場合being debuggedと表示し、そうでない場合はnot being debuggedと表示します。

```
#include <stdio.h>
#include <signal.h>
int being_debugged = 1;
void signal_handler(int signum) {
    being_debugged = 0;
}

int main() {
    signal(SIGTRAP, signal_handler);
    __asm__ __volatile__("int3");
    if (being_debugged) {
        printf("being debugged\n");
    } else {
        printf("not being debugged\n");
    }
    return(0);
}
```

実行結果は以下のようになります。

```
% gcc test.c
% ./a.out
not being debugged
% gdb ./a.out
... 略 ...
(gdb) run
Starting program: /tmp/a.out
Program received signal SIGTRAP, Trace/breakpoint trap.
```

```
0x080483a9 in main ()
(gdb) c
Continuing.
being debugged
```

これは悪意のあるプログラムにおいて、デバッガ上で実行されるのを阻止するために使われるテクニックの1つです。詳しくは参考文献『セキュリティウォリア』を参照してください。

まとめ

上で述べたようなことをしたい場面はそれほど多くないと思いますが、実行時の特定のタイミングで処理を中断してデバッガで状態を見たり、マクロの中などデバッガでブレークポイントを設定しにくいところを調べるのに便利なのではないかと思います。

参考文献

- 『セキュリティウォリア』(Cyrus Peikari、Anton Chuvakin 著、西原啓輔監訳、伊藤真浩、岸信之、進藤成純訳、オライリー・ジャパン)

—— Satoru Takabayashi

7章 その他の Hack
Hack #93-100

本章では、これまでの章のカテゴリに収まらないさまざまなHackを紹介します。最後に、文献案内として今後の Binary Hack の手引きとなる書籍や Web サイトなどを紹介します。

HACK #93 Boehm GC の仕組み

Boehm GCは、C/C++言語にガベージコレクションを導入するためのメモリ管理ライブラリです。

Boehm-Demers-Weiser Conservative Garbage Collector（Boehm GC）は、C/C++ 言語にガベージコレクション（GC）を導入できる強力なメモリ管理ライブラリです。多くの UNIX、Windows、OS/2 に対応し、マルチスレッド動作もサポートしています。また w3m や gcj などにも採用され、動作実績の多い「堅い」ライブラリでもあります。

Boehm GC の使い方

Boehm GC は開発者である Boehm 氏の Web サイト（http://www.hpl.hp.com/personal/Hans_Boehm/gc/）からソースコードで配布されています。執筆時点での最新版は 6.6 です。

基本

Boehm GC を使用するには gc.h をインクルードし、メインプロセス（main 関数の最初がよい）の中で初期化関数 GC_INIT を呼び出します。

```
#include "gc.h"

int main(int argc, char** argv) {
  GC_INIT();
```

後は C プログラムの中にある malloc/free を Boehm GC 提供の malloc に置き換えるだけです。Boehm GC が管理するヒープメモリがいっぱいになると自動的に GC が起動され、使用されていないゴミオブジェクト（ガベージ）が回収されます。

元の関数	置き換える関数
malloc	GC_ malloc
calloc	GC_calloc
realloc	GC_realloc
free	GC_free

GC_mallocの高速版としてGC_malloc_atomicが用意されています。これは整数型の配列など内部にポインタを含まないオブジェクト用です。GC_freeは通常は何の処理も行わない関数ですので、freeは削除しても OK です。

ファイナライザ(Finalizer)

GC_mallocなどによって割り付けられたオブジェクトは、回収直前のタイミングで呼び出されるコールバック関数を登録することが可能です。このような関数をファイナライザ(Finalizer)と呼びます。ファイナライザとして登録できるのは以下のプロトタイプを持つ関数です。

```
typedef void (*GC_finalization_proc)(GC_PTR obj, GC_PTR client_data);
```

登録にはGC_register_finalizerを使います。オブジェクトobjにファイナライザfnが結び付けられます。登録時にclient_dataで渡したデータを、ファイナライザ時に受け取ることも可能です。1つのオブジェクトに割り当て可能なファイナライザは1つだけなので、過去に登録されたものがold_fnとold_client_dataで引き渡されます。

```
void GC_register_finalizer(GC_PTR obj,
                           GC_finalization_proc fn, GC_PTR client_data,
                           GC_finalization_proc* old_fn, GC_PTR* old_client_data);
```

GC_register_finalizerをfnをNULLで呼び出すと、実行の途中でもファイナライザを解除することが可能です。なおファイナライザを使うには、いくつか注意事項があります。

- ファイナライザが呼び出された後は、強制的にオブジェクトが回収されます。Javaのファイナライザと異なり「オブジェクトの復活」はありません。ファイナライザの中でオブジェクトのポインタを保存するとダングリンポインタができるので大変危険です。
- ファイナライザに登録されたオブジェクトに親子関係がある場合、親オブジェクトから先にファイナライザが呼び出されます。そのためファイナライザ登録されたオブジェクト同士に循環参照があると、警告メッセージが表示されファイナライザが呼び

出せません。循環参照を作る可能性のあるオブジェクトのファイナライザ登録には GC_register_finalizer_ignore_self、GC_register_finalizer_no_order を使ってください。

- マルチスレッドプログラムでは、GCを引き起こしたスレッドがファイナライザの実行を行います。そのためどのスレッドがファイナライザを呼び出しても問題が起きないように実装する必要があります。

マルチスレッドで使う

マルチスレッド環境下でBoehm GCを使うには、プラットフォームごとの制限を守る必要があります。Linux の場合には以下のような制限があります。

- すべてのモジュールの先頭で gc.h をインクルードし、-DGC_LINUX_THREADS と -D_REENTRANT を指定してコンパイルする必要があります。これは pthread ライブラリ関数を Boehm GC のラッパー関数と置き換えるためです。
- GC 中に平行して dlopen 系の関数を使うと、動作が保証されません。
- Thread Local Storageを使うとガベージでないオブジェクトを誤回収する危険性があります。

Boehm GC の仕組み

Boehm GCはMark-Sweepアルゴリズムを用いています。GC_mallocが新しいオブジェクトの確保に失敗すると、GC が起動して以下の処理を行います。

1. GC ルートから参照されている Boehm GC のオブジェクトは、生きているとみなしてマーキングを行います。GCルートはグローバル変数、ローカル変数、mallocで確保したメモリなど、Boehm GC 管理外のメモリすべてです。
2. マーキングされたオブジェクトの中にもポインタが含まれています。これを再帰的にたどってオブジェクトをマーキングして行きます。最終的にマーキングされずに残ったオブジェクトは、どこからも参照されていないオブジェクト(ガベージ)と判断できます。
3. ガベージが占めているメモリ領域をフリーリストにつなげることでメモリ回収が完了します。

しかし実際に上の処理を行うには、さまざまな Hack が必要なのです。

GC ルートをかき集める

まずGCルートとなるデータは、レジスタ、スタック、データセグメント、ヒープなどに分散しています。レジスタ内のデータは、setjmpの動作を模擬するような「ネイティブ」コードで内容をメモリに吐き出します。

スタックは、スタックボトム(スタックポインタの初期位置)から現在のスタックポインタまでをGCルートに加えます。現在のスタックポインタはローカル変数のアドレスから推測できます。

プロセスが共有ライブラリをロードしている場合、メモリ空間の中にテキストセグメント、データセグメントがまだらに入り乱れることになります。データ部分だけをGCルートに加えるためには、共有ライブラリのマップ情報が必要です。Boehm GCでは「**[Hack #65] ロードしている共有ライブラリをチェックする**」と同様の手法で、プロセス内のメモリを調査しています。例えばLinuxではdl_iterate_phdr(3)に以下のようなコールバック関数を与えてGCルートを得ることができます。

```
#include <stddef.h> /* for offsetof macro */
#include <link.h>
#include <elf.h>

int GC_register_dynlib_callback(struct dl_phdr_info* info, size_t size, void* ptr) {
  int i;
  const ElfW(Phdr)* p = info->dlpi_phdr;

  for (i = 0; i < (int)(info->dlpi_phnum); ((i++),(p++)))
    switch (p->p_type) {
      case PT_LOAD: {
        char * start;
        if (!(p->p_flags & PF_W)) break;
        start = ((char *)(p->p_vaddr)) + info->dlpi_addr;
        /* [start, start + p->p_memsz) のメモリ領域をルートセグメントに加える */
      } break;
    }

  return 0;
}
```

malloc(3)でとったメモリやmmap/munmap(2)で確保したメモリもGCルートになります。後者でとられたメモリはページが割り当てられていない「穴」が存在する危険性があります。これは「**[Hack #81] SIGSEGVを使ってアドレスの有効性を確認する**」のようにSIGSEGVを捕捉することでメモリページの状態を把握しています。

保守的なGC

OSの助けを借りてデータの置かれている領域の同定ができるのですが、C/C++のプログ

ラムの場合、あるデータがポインタ型なのか整数型なのか区別がつきません。Boehm GCはメモリ領域をワード単位で探索し、ポインタに見えるビット列は全部ポインタとして扱います。このようにオブジェクトを指しているポインタを正確に同定できない / しない GC を“Conservative（保守的な）GC”と呼びます。逆にどこがポインタでどこがデータなのか正確に区別できる GC を“Precise（正確な）GC”、“Type-accurate（型正確な）GC”、“Exact（厳格な）GC”などと呼びます。

Conservative GC は Precise GC と比べて 2 つの点で不利です。1 つは Conservative GC はビット列を大雑把にポインタと見なすために、ガベージの一部が回収されない可能性があり、メモリ効率が少し悪くなることです。もう1つはPrecise GCでは任意のオブジェクトを移動しポインタを張り替えることでヒープメモリ中の空き領域を詰める整列（コンパクション）が可能なのですが、Conservative GCではそれができないという点です。そのためプログラムが進むとメモリの空き領域の断片化が問題になります。

ただ、C/C++言語でPrecise GCを行うには最低でもコンパイラ、リンカのサポートが必要になり、既存のライブラリとリンクもできなくなります。過去の資産を生かすためにも、Boehm GC は Conservative GC を採用しています。

スレッド停止、再開の機能がない OS でスレッドを停止させる

マルチスレッド動作時にはGC_mallocを呼び出しヒープメモリの枯渇に気づいたスレッドがGCを実行します。GC中にGCを実行しているスレッド以外が動いていると、オブジェクトの内容を書き換えられマーキングが正しく行えなくなるため、"stop the world"と他のスレッドをすべて停止させる必要があります。

ところでPthreadライブラリはスレッドの停止、再開機能が定義されていません。pthread_suspend_npなどを処理系独自のAPIとして定義しているOSもありますが、Linux などにはスレッドの停止、再開機能がそもそも存在しません。そこでBoehm GCはシグナルを使った擬似スレッド停止、再開機能を実装しています。Linux ではSIGPWR、SIGXCPUシグナルがあまり使われていませんのでこれを転用します。

```
#define SIG_THR_SUSPEND  SIGPWR
#define SIG_THR_RESTART  SIGXCPU

__thread int count; /* サスペンドの多重度を記録 */
sigset_t  suspend_handler_mask;

void init() {
  sigfillset(&suspend_handler_mask);
  /* 待機中にも受け付けるシグナル */
  sigdelset(&suspend_handler_mask, SIG_THR_SUSPEND);
  sigdelset(&suspend_handler_mask, SIG_THR_RESTART);
  sigdelset(&suspend_handler_mask, SIGINT);
```

```
  sigdelset(&suspend_handler_mask, SIGQUIT);
  sigdelset(&suspend_handler_mask, SIGABRT);
  sigdelset(&suspend_handler_mask, SIGTERM);
}
```

スレッドごとにSIG_THR_SUSPEND、SIG_THR_RESTARTシグナルのハンドラとしてthread_suspender_handler を登録しておきます。シグナルハンドラ内ではsigsuspend(2)で待機します。

```
void thread_suspender_handler(int sig, siginfo_t* sig_info, void* sig_data) {

  if (sig == SIG_THR_SUSPEND) {
    count++;
    if (count == 0) {
      do {
        sigsuspend(&suspend_handler_mask);
      } while(count > 0);
    }
  } else if (sig == SIG_THR_RESTART) {
    count = 0;
  }
}
```

後はGCスレッドから他のスレッドに対してpthread_kill(3)で要求を伝えます。

```
pthread_kill(target_thread, SIG_THR_SUSPEND);
pthread_kill(target_thread, SIG_THR_RESTART);
```

まとめ

Boehm GCの使い方と簡単な原理を紹介しました。C/C++プログラムで、特にマルチスレッド環境でのGCはかなり無理がある処理です。だからこそBoehm GCの内部実装はHackの“嵐”になっています。プラットフォームに深く依存したメモリ管理のHackに興味のある方は、Boehm GCのソースコードをぜひ参照してください。

参考文献

- 「Richard Jones's Garbage Collection Page」（http://www.ukc.ac.uk/computer_science/Html/Jones/gc.html）
  GCの教科書の決定版と言えるページ。

—— Minoru Nakamura

HACK #94 プロセッサのメモリオーダリングに注意

本Hackでは、メモリアクセスの順序を決めるメモリオーダリングについて解説します。メモリオーダリングが原因のバグは最も調査が難しい部類に属します。

“メモリオーダリング(memory ordering)”あるいは“メモリコンシステンシ(memory consistency)”はプロセッサがメモリアクセス命令をどのような順序で発行するかのルールです。現代のプロセッサのほとんどはパフォーマンス向上のために命令の並び順とは異なる順序でメモリアクセスすることを認めています。マルチプロセッサシステムで並列、非同期プログラムを作成したり、ドライバの開発を行う際にはメモリオーダリングを意識しましょう。

このプログラムのどこにバグがあるのでしょうか？

次のコードは2つのスレッド間でクリティカルセクションを保護するロックのプログラムです。スレッドはそれぞれ0と1の番号を持っており、クリティカルセクションに進入したいスレッドは、まずlock_array[me]を1にして(8行目)、その後でもう1つのスレッドのlock_array[other]を確認します(9行目)。他のスレッドが同時進入を試みていれば、1が立っていますので競合が発生です。ロックは失敗ですので、自分のlock_array[me]を0に戻してリトライします。

このプログラムは一見うまく動きそうですが、動かしているプロセッサの種類によっては2つのスレッドがクリティカルセクションに同時侵入する危険性があります。

```
1  volatile int lock_array[2];
2
3  void routine(int me /* 0 or 1 */) {
4    int other = 1 - me;
5
6  start_point:
7    /* lock */
8    lock_array[me] = 1;
9    if (lock_array[other] != 0) {
10      lock_array[me] = 0;
11       /* wait a moment */
12       goto start_point;
13   }
14
15   /* critical section */
16
17   /* unlock */
18   lock_array[me] = 0;
19 }
```

原因は8行目と9行目の実行順序にあります。volatile修飾子によって、lock_array[me]のストア命令→lock_array[other]のロード命令の順に機械語化されることは保証されています

が、プロセッサの内部の機構がプログラムの順序を破ってメモリアクセスを入れ替えてしまうことがあるのです。

メモリアクセスが逆転すると、相手のステータスを見てから自分のステータスを変えるプログラムになり、動作は保証されません。このような意図しない事態が起きるのがメモリオーダリング問題の難しさです。

プロセッサのメモリアクセス順序

メモリオーダリングは、同一アドレスに対するアクセス順序と異なるアドレスに対するアクセス順序の2種類があります。

同一アドレスに対するメモリアクセスは、ほとんどのプロセッサでプログラム通り実行されることが保証されています。しかしIA-64ではこの順序も守られず、下のコードを実行するとThread1の4のロード命令が3のロード命令を追い越してa = 2、b = 1という結果が出ても許されるのです。

```
 Thread1     Thread2
==========  ==========
1: *p = 1;
2:          *p = 2;
3: a = *p;
4: b = *p;
```

一方、異なるメモリアドレスに対するメモリアクセスの順序は、現行のプロセッサのほとんどで問題になります。

メモリアクセスは、Write After Read（WAR）、Read After Write（RAW）、Read After Read（RAR）、Write After Write（WAW）の4種類のパターンに分類されます。例えばWARはストア命令の後にロード命令が行われるパターンです。すべてのパターンで順序が守られるメモリオーダリングを“Strong Ordering”と呼びます（Sequential Ordering、またはProgram Orderingと呼ぶこともある）。逆にすべてパターンで順序の入れ替えを認めるメモリオーダリングを“Weak Ordering”と呼びます（Relaxed Memory Orderingと呼ぶこともある）。Strong orderingとWeak orderingの間でさまざまにメモリオーダリングがありますが、表7-1に主なメモリオーダリングとそれを採用しているプロセッサをまとめました（順序が保証されるものは○、逆転の可能性があるものは×。ただし一般に使われない特殊なモードや、一部の命令だけで有効なメモリオーダリングは除外しています）。

表 7-1　メモリオーダリングの種類

種類	RAR	WAR	WAW	RAW	採用しているプロセッサ
Strong Ordering	○	○	○	○	i386、PA-RISC
Total Store Ordering	○	○	○	×	IBM 370、SPARC（ノーマル）、i486、Pentium
Partial Store Ordering	○	○	×	×	SPARC（PSO モード）
Speculative Processor Ordering	×	×	○	×	PentiumPro、Pentium 4
Weak Ordering	×	×	×	×	IA-64、Alpha、POWER、SPARC（RMO モード）

メモリバリア命令

メモリアクセスの逆転が起きるのは、プロセッサとキャッシュの間にある“ストアバッファ”と呼ばれる機構が原因です。ストアバッファはプロセッサの中にある一種の「キャッシュ」で、数命令分のストア命令をプロセッサの中で待機させることができます。その結果、以下のような効果が得られます。

- 同じアドレスにストア命令が2回実行される場合、最初のストア命令がキャッシュへ書き込むのをキャンセルします。
- 同じアドレスにストア命令→ロード命令が連続する場合、後発のロード命令はストアバッファからデータをもらいキャッシュの読み込みをパスします。

ストアバッファによってキャッシュへのアクセスを減らせるため、高速化が可能なのですが、異なるアドレスに対するメモリアクセスがゆるくなります。そのためプロセッサは“メモリバリア命令”あるいは“フェンス命令”と呼ばれるメモリアクセスをシリアライズ（順序化）する専用命令を用意しています。

メモリバリア命令の形式と効果はアーキテクチャによって異なるのですが、x86 の場合に表 7-2 のような命令が用意されています。

表 7-2　x86 のメモリバリア命令

命令	採用されたプロセッサ	概要	順序化できる依存
sfence	Pentium III	ストア命令のシリアル化	WAW
lfence	Pentium 4	ロード命令のシリアル化	RAR
mfence	Pentium 4	ロード命令とストア命令のシリアル化	4 パターンすべて

GCCでは以下のインラインアセンブラのマクロでコード中にメモリバリア命令を埋め込むことができます。

```
#define membar()  asm volatile("mfence":::"memory")
```

前のサンプルプログラムは8行目と9行目の間にこのmembarを挟み込むことでエラーがでなくなります。主なプロセッサアーキテクチャのメモリバリア命令を表7-3にまとめます[1]。

表7-3 さまざまなアーキテクチャのメモリバリア命令

アーキテクチャ	命令	備考
x86	sfence、lfence、mfence	
IA-64	mf	†
PowerPC & POWER	sync、lwsync、eieio、isync	
SPARC	stbar、membar	
PA-RISC	sync	††
Alpha	mb、wmb	
MIPS	sync	*

† IA-64はリリースアクワイヤセマンティックスに基づく独特なメモリバリアセマンティクスが他にあり、そちらが主に使われます。

†† PA-RISCは基本的なメモリアクセスはstrong orderingですが、キャッシュ制御命令などが順序性が弱くなっています。

* 著者が調べた範囲では、MIPSはアーキテクチャレベルではメモリオーダリングが決まっていないようです。メモリバリアを使うかどうかはチップやシステム構成によって異なるようです。

アトミック命令によるシリアライズ効果

ほとんどのプロセッサはCompare-And-Swap命令などのアトミック命令を発行した場合に、メモリアクセスがシリアライズされる副作用があります。x86の場合はlockプリフィックスの付いたアトミック操作は、「先行する命令が完了しストアバッファが空になるまで待つ」と定義されていますのでメモリバリア命令よりも強力にシリアライズを行います。

ただしアトミック命令によるシリアライズ効果はすべてのアーキテクチャで得られるわけでありません。IA-64、Alpha、POWERではアトミック操作対象のアドレス以外のメモリアクセスは、アトミック命令を越えて順序を入れ替えることが許されています。

またシステムに与える影響についても注意が必要です。メモリバリア命令はプロセッサ内の命令実行順序をシリアライズするだけの命令ですが、アトミック命令はシステムバス、メモリバスをロックするため他のプロセッサに影響を与える重い命令です。そのためメモリバリア命令で済む場合には、極力メモリバリア命令を使った方が効率がよいでしょう。

プログラム言語や API によるサポート

理想的にはC/C++言語レベルでメモリバリアやアトミック命令を使うことができればよいのですが、2006年現在では処理系に強く依存する方法(インラインアセンブラ、組み込み関数)しかありません。メモリバリアの標準的なインターフェイスを定めようとする試みや、Javaのように言語仕様でメモリモデルを定義しようとする試みが進められています。今後の展開に期待しましょう[3,4]。

また、OSやシステムライブラリが提供するAPIの中には、仕様としてメモリアクセスのシリアライズ効果を保証するものがあります[2]。たとえばpthread_mutex_lockは、バリア命令同様にAPI呼び出し前後のメモリアクセスをシリアライズすることができます。高コストなためメモリバリアの替わりには使えませんが、このようなAPIが呼ばれる前後ではメモリオーダリングを切り離して考えることができます。

まとめ

メモリアクセスの順序を決めるメモリオーダリングについて解説しました。メモリオーダリングの問題は、マルチスレッドプログラムを書く人でもなかなか意識できません。しかし著者の経験では、メモリオーダリングが原因のバグは最も調査が難しい部類に属します。思わぬ落とし穴に落ちぬよう気をつけましょう。

参考文献

1 「Linux Journal, Memory Ordering in Modern Microprocessors」(Part I：http://www.linuxjournal.com/article/8211、Part II：http://www.linuxjournal.com/article/8212)
2 「The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 4.10 Memory Synchronization」(http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap04.html#tag_04_10)
3 「Threads and memory model for C++（Hans Boehm)」(http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/)
4 「Java Community Process, JSR-000133 Java Memory Model and Thread Specification Revision」(http://jcp.org/aboutJava/communityprocess/final/jsr133/index.html)

—— Minoru Nakamura

HACK #95 Portable Coroutine Library(PCL)で軽量な並行処理を行う

本Hackでは、PCLを用いてC言語でコルーチンを使用する方法と、PCLの原理を解説します。

本Hackでは、軽量なスレッドであるコルーチンの解説と、Cなどの言語でコルーチンを実現するPortable Coroutine Libraryの解説を行います。

コルーチンとは

コルーチン(co-routine)とは、マイクロスレッド(micro-thread)やファイバー(fiber)とも呼ばれます。fiber(繊維)はthread(糸)と対応させた名前で、筆者はなかなかに気が効いている命名であると思います。

定義としてはスレッドとの対比で、「非時分割で非プリエンプティブ(協調型)なスレッド」「切り替えを自分で(できる／しなければならない)スレッド」などと言った説明がわかりやすいと思います。サブルーチンとの対比で考えると、サブルーチンはsubという接頭辞が表すように、呼び出し側が親となってそちらに実行が帰りますが、コルーチンでは接頭辞coの通り呼び出し側も呼ばれ側のどちらも協調して対等に動作します。

コルーチンを仮想コードで書いた実行イメージとしては以下のようになります。

```
coroutine() {
  print("1\n")
  yield
  print("3\n")
}

main() {
  coroutine()
  print("2\n")
  yield
  print("4\n")
}
```

このコードの場合、1234が順番に出力されるでしょう。上記コードはyieldによって、別なコルーチンに処理を渡しているのが特徴的です。スレッドと異なり「切り替えを自分で(できる／しなければならない)」ことがわかると思います。

このように、コルーチンはスレッドのように同時実行されていないため、同期処理などは不要ですし、軽量です。スレッドよりも便利なシーンも多いと思います。筆者はゲームの敵キャラクタの動作の処理の部分などによく利用しています。

Portable Coroutine Library (PCL)

コルーチンの実現には、スタックと実行コンテキストとして使用されているレジスタの退避が必須であり、通常言語システム自体がサポートしていないかぎりは使用することができません。しかし、アーキテクチャの差異を吸収するPortable Coroutine Library（PCL）を使用すれば、コルーチンが採用されていない言語でもポータブルにコルーチンを使用することができます。

PCLはhttp://xmailserver.org/libpcl.htmlから、GPL2をもとに配布されており、執筆時点での最新版は1.6です。同種のライブラリとしてはこちらはかなりサイズが大きいライブラリですが、GNU Pth（http://www.gnu.org/software/pth/）も有名です。また、C++では、Boost Coroutine（https://boost_consulting.com:8443/trac/soc/wiki/coroutine）というプロジェクトも最近始まっています。

先ほどの例をPCLを用いて書き直したサンプルを以下に示します。

```
#include <pcl.h>
#include <stdio.h>

#define CO_STACK_SIZE (32 * 1024)

void spawn(void *arg) {
    printf("%d\n", 1);
    co_resume();
    printf("%d\n", 3);
}

int main() {
    coroutine_t co;

    co = co_create(&spawn, NULL, NULL, CO_STACK_SIZE);
    co_call(co);
    printf("%d\n", 2);
    co_call(co);
    printf("%d\n", 4);

     return 0;
}
```

上記も実行すると1234の順で表示されます。ご覧のように、co_callでコルーチンを実行させ、co_resumeでコルーチンから復帰しています。co_createの第2引数は、第1引数の関数を呼ぶ時の引数として用いられますが、ここでは使用していません。

PCLの実装

pclのコードはほとんどpcl.cの1つで500行程度と、非常に短くなっていて、基本的動作

としては、スタックを動的確保した空間に保存し、レジスタを調整してsetjmp/longjmpしているだけです。この実装の多くはhttp://www.gnu.org/software/pth/rse-pmt.psの論文で解説されています。

まず、ucontext(3)という最近のUNIXに存在する機構があれば、それを利用します。これはまさにコルーチンを行うための機構です。makecontextでコンテキストを生成し、getcontextやswapcontextを使用してコンテキストの退避、変更を行います。

これがない環境ではコンテキストの移動はsetjmp/longjmpで行い、なんらかの方法でスタックをもう1つ作ってコンテキストを保存することになります。

2つ目の方法はsigstack(2)/sigaltstack(2)というシステムコールを使用するトリックです。このアルゴリズムは上記論文で詳しく述べられていますが、新しいシグナルスタックをsigstack/sigaltstackでシステムに通知した後、いったん自分にシグナルを撃ってシグナルハンドラ内でsetjmpして、もう1つコンテキストを作り、戻ったらスタックを元に戻す、というのが概要です。それ以降はsetjmp/longjmpで行き来することができます。

最後に泥くさいですが確実な方法は、スタックを自前で動的確保してアーキテクチャやOSごとに必要な空間をコピーして保存し、jmp_buf構造体の内部実装に踏み込んでスタックを指すレジスタとプログラム実行位置を指すレジスタを保持する空間を自力で書き換えることです。この方法は確実ですが環境依存のため、激しくifdefで区切られています。

PCLの対応環境を増やす場合は最後の方法のifdefを増やすのが簡単でしょう。例えば筆者はmingw32環境でPCLを使用したかったため、Ioというコルーチンをサポートしているスクリプト言語のScheduler.cから該当個所をコピーアンドペーストして使用しています。筆者の作成したPCL-1.6へのパッチはhttp://shinh.skr.jp/binary/pcl_ioarch_1_6.patchにあります。

まとめ

本Hackでは、PCLを用いてC言語でコルーチンを使用する方法と、PCLの原理を紹介しました。

—— Shinichiro Hamaji

HACK #96 CPUのクロック数をカウントする

CPUのクロックを読み出すことにより、細かい操作にかかる時間を測定することができます。

最近のプロセッサの多くにはCPUのクロックやそれに類するクロックのカウンタが入っており、その値をソフトウェアから得ることができます。この値は、ロックなど、非常に小さな処理の速度を測定するときに有用です。また、乱数の種や、ブートしてからの時間を知ることに使える場合もあります。さらに、1秒間にいくつクロックが進むかを調べれば周波数

を測定することもできます。

さまざまなプロセッサ

以下のプロセッサにはクロックを読み出す機能があります。

プロセッサ	命令	ビット数	備考
Pentium	`rdtsc`	64	EDXに上位32ビット、EAXに下位32ビットが格納される
Itanium	`mov R = ar44`	64	Rに64ビット格納される
32ビットPowerPC	`mftbu H、mftb L`	64	Hに上位32ビット、Lに下位32ビットが格納される †、††
64ビットPowerPC	`mftb R`	64	Rに64ビット格納される
UltraSPARC	`rd %tick, R`	64	Rに64ビット格納される *
HP PA-RISC 2.0	`mfctl %cr16, R`	64	Rに64ビット格納される
HP PA-RISC 1.0	`mfctl %cr16, R`	32	Rに32ビット格納される
Alpha	`rpcc R`	32	Rの下位32ビットに32ビット格納される **
S/390	`stck ADDRESS`	64	指定したアドレスから8バイトに格納される
MIPS	`mfc0 R, $9`	32	‡
SH64	`getcon cr62, R`		‡

† 32ビットPowerPCでは2つの命令が必要で、64ビットの値を一度に読み出すことができないため、`mftbu`、`mftb`、`mftbu`の順に読み出し、2つの`mftbu`で読み出した上位32ビットが変化していないことを確認し、変化していたらやり直す必要があります。

†† PowerPCで得られるクロックはCPUのクロックではなくバスのクロックの1/4です。

* UltraSPARCで64ビットレジスタを扱うにはSPARC-V8+ ABI以上である必要があります。

** Alphaのレジスタは64ビットで、上位32ビットには、OS依存な値が入ります。Linuxの場合には上位32ビットと下位32ビットを加えるとプロセス単位のクロック数になるような補正値が入ります。

‡ 未テスト、未確認。

実装

ここでは、例としてPentium、AMD64、32ビットPowerPCの実装を示します。

Pentium

Pentiumおよび Pentium互換のプロセッサでは以下のように実装できます。

```
unsigned long long clockcount_pentium(void)
{
  unsigned long long ret;
  __asm__ volatile ("rdtsc" : "=A" (ret));
  return ret;
}
```

ここでrdtscはEDX:EAX(EDXに上位32ビット、EAXに下位32ビットを入れて64ビット値を表す方法)に64ビット値を格納しますが、EDX:EAXというのは、Pentiumにおいて64ビット値を表現する標準的な方法です。そのため「**[Hack #23] GCCでインラインアセンブラを使う**」でも述べられているconstraintでもEDX:EAXを示す"A"という指定が可能で、ここではそれを使用して、値を変数に取り出しています。

AMD64

AMD64の命令セットはPentiumの上位互換ですからrdtscが使用でき、以下のように実装できます。

```
unsigned long long clockcount_amd64(void)
{
  unsigned int eax, edx;
  __asm__ volatile ("rdtsc" : "=a" (eax), "=d" (edx));
  return eax | (unsigned long long)edx << 32;
}
```

ここではrdtscでEDX:EAXに取り出したクロックを、"a"と"d"というEAX、EDXに対する指定を使って個々の変数に取り出し、Cのレベルで64ビットの値に構成し直しています。

なお、この実装はAMD64だけでなくPentiumでも使用できます。逆に、Pentiumの項で示した実装はAMD64では使えません。これは、"A"という指定は、AMD64ではRAXを示すため、EDX:EAXを取り出すことには使えないからです。

32ビットPowerPC

32ビットPowerPCでは以下のように実装できます。

```
unsigned long long clockcount_powerpc32(void)
{
  unsigned int h1, h2, l;
  do {
    __asm__ volatile ("mftbu %0" : "=r" (h1));
    __asm__ volatile ("mftb %0" : "=r" (l));
    __asm__ volatile ("mftbu %0" : "=r" (h2));
  } while (__builtin_expect(h1 != h2, 0));
  return (unsigned long long)h1 << 32 | l;
}
```

ここではmftbuで上位32ビット、mftbで下位32ビットを得ています。しかし、最初のmftbuとmftbの間に、クロックが進んで下位32ビットがあふれ、上位32ビットがインクリメントされてしまうかも知れません。そこで、mftbuをもう一度行い、上位32ビットが変化していたら最初からやり直すようにしています。また、やりなおしが必要な可能性は非常に低いため、__builtin_expectにより、gccにh1 != h2が高確率で0になることを教え、分岐予測を助けています。

周波数とブートからの経過時間を測定する

上記のclockcount_pentium()を使用して、クロックの周波数とブートからの経過時間を測定するには次のようなプログラムで可能です。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

int main()
{
  int ret;
  struct timeval tv1, tv2;
  unsigned long long c1, c2;
  double t, speed, uptime;

  c1 = clockcount_pentium();
  ret = gettimeofday(&tv1, NULL);
  if (ret == -1) { perror("gettimeofday"); exit(1); }
  for (;;) {
    sleep(1);
    c2 = clockcount_pentium();
    ret = gettimeofday(&tv2, NULL);
    if (ret == -1) { perror("gettimeofday"); exit(1); }
    t = tv2.tv_sec - tv1.tv_sec + (tv2.tv_usec - tv1.tv_usec) * 1e-6;
    speed = (c2 - c1) / t;
    uptime = c2 / speed;
    printf("0x%llx %lf[Hz] %lf[sec]\n", c2, speed, uptime);
  }
}
```

このプログラムの実行結果はたとえば次のようになります。

```
0xa0ed80a7f9f7 1261795267.448678[Hz] 140230.297854[sec]
0xa0edcdb8978f 1274756551.617955[Hz] 138805.498897[sec]
0xa0ee1bb2474b 1284146489.383961[Hz] 137791.543871[sec]
0xa0ee66e312cf 1277204231.704936[Hz] 138541.499022[sec]
...
```

この結果から、周波数が1.2～1.3GHzであり、ブートから約38時間ほど経過していることがわかります。

注意

周期

カウンタが32ビットの場合、仮に周波数が1GHzだとすると、カウンタは約4秒でオーバーフローします。そのため、小さな処理の処理時間を測定するのには問題ありませんが、ブートからの時間などの長い時間を知ることはできません。

SMP

マルチプロセッサの場合、プロセッサごとにカウンタが存在することがあります。この場合、各カウンタは完全に同期しているとはかぎりません。初期状態で同期させたとしても、だんだんとずれることもあり得ます。

クロックの変化

最近のプロセッサでは電力節約のために動作周波数を低下させることがあります(例えば、IntelのSpeedStep、AMDのPowerNow!など)。この場合、本Hackで読み出したクロックは、必ずしも実際の速度と一定の比にはなりません。

まとめ

プロセッサのクロックを読み出すことにより、細かい操作にかかる時間を測定することができます。

—— Akira Tanaka

HACK #97 浮動小数点数のビット列表現

計算機の中ではIEEE754といった規格に従った形式で、実数の近似値がビット列として表現されています。

本Hackでは、計算機の中で実数(の近似値)がどのように表現されているかを説明します。

浮動小数点数

計算機で実数を扱いたい場合、浮動小数点数(floating-point number)で近似して表現することが一般的です。浮動小数点数についてはIEEE754という規格があり、近年ではほぼすべてのプロセッサがこの規格に準拠しています。IEEE754規格は、数値をどのようにビット列で

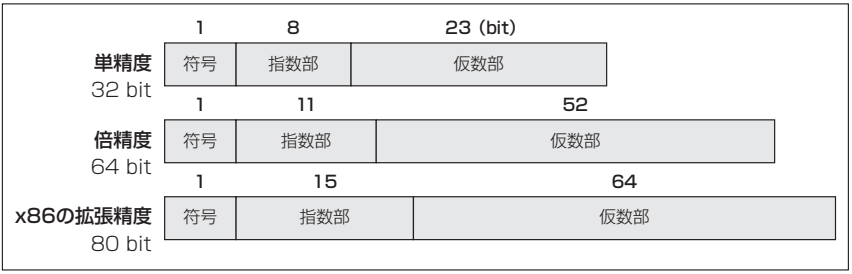


図 7-1　浮動小数点数の x86 での表現形式

表現するかに始まって、丸めの方向制御、例外、あふれ時の挙動などを規定しています。

IEEE754では、数値の表現形式として、32ビットで表現される単精度(single precision)数、64ビットの倍精度(double precision)数、また、両者を元にビット数を増やした拡張単精度、拡張倍精度を規定しています。C言語では、float型が単精度、double型が倍精度、long double型が拡張倍精度です。

多くのプロセッサは単精度と倍精度をサポートしており、プロセッサによっては64ビットより長い(例えば128ビットの)拡張倍精度もサポートします。例えばx86プロセッサの拡張倍精度は80ビットです。逆に、多くのGPU (Graphics Processing Unit) のように、単精度で事足りることを理由として倍精度以上をサポートしないプロセッサも数多くあります。

浮動小数点数は、符号(sign)、指数部(exponent)、仮数部(significand、mantissa)の3つから成ります。この3つを合わせて、単精度では32ビット、倍精度では64ビットとなります。符号、指数部、仮数部それぞれのビット列をそれぞれs、e、fとした場合、そのビット列は、基本的には次の値を表します。

単精度:　$(-1)^s\ 2^{e-127}\ 1\cdot f$
倍精度:　$(-1)^s\ 2^{e-1023}\ 1\cdot f$

ここで"・"は小数点です。例えば、

```
1 01111111 11000000000000000000000 (2進数)
```

この単精度数は、以下の通り -1.75 を表します。

$(-1)^1\ 2^{127-127}\ 1\cdot 11 = -1.75$

小数1・11は2進数であることに注意してください。指数部のビットがすべて0、または、すべて1の場合には特別な規定があり、無限大やNaN、非正規数など、また別の種類の数値を表します。

ビット列から浮動小数点数を作る

上記のビット列が-1.75であることを確認します。以下では、int型とfloat型のサイズが両者とも32ビットであることを仮定します。この仮定は、32ビット/64ビットのUNIX系OSとWindowsのほとんどで成り立ちます。

```
unsigned int i;
float f;

i = 0xbfe00000u;  /* -1.75 */
printf("%x\n", i);

f = *((float *)&i);
printf("%g\n", f);
```

このコードを実行すると以下の通り表示され、前掲のビット列(0xbfe00000)が-1.75を表すことが確認できます。

```
bfe00000
-1.75
```

Javaでは、このように同一のメモリ領域を複数の異なる型で扱うことはできないようになっています。その代わり、ビット列から浮動小数点数への変換を行うためのFloat#intBitsToFloat、Double#longBitsToDoubleメソッドが用意されています。

まとめ

計算機の中ではIEEE754といった規格に従った形式で、実数の近似値がビット列として表現されています。

—— Kazuyuki Shudo

HACK #98 x86が持つ浮動小数点演算命令の特殊性

本Hackでは、他のプロセッサとは異なる演算結果を導くことすらあるx86 FPUの特殊性を説明し、その回避手法を紹介します。

x86プロセッサの浮動小数点演算器(FPU)には、IEEE754規格準拠であるにもかかわらず、他のプロセッサとは若干異なる点があります。これによって、四則演算の結果が他のプロセッサと食い違う場合もあります。

特殊な点

x86プロセッサの浮動小数点(以下FP)演算命令にはいくつか特殊な点があります。近年、FPレジスタも整数レジスタと同様に汎用のレジスタが並んでいるというアーキテクチャが一般的ですが、x86のFPレジスタはスタックになっていて、スタックトップは特別な役割を持ちます。

しかしx86の特殊性はこれだけではありません。IEEE754規格に準拠しているにもかかわらず、FPレジスタ上の数値の表現が他の多くのプロセッサとは異なり、それが理由で演算結果が他のプロセッサとは変わる場合もあります。

例えば、次のプログラムの実行結果が、他のIEEE754準拠プロセッサとは違うものになります。

```
#include <stdio.h>

double dmul(double a, double b) {
  return a * b;
}

int main(int argc, char **argv) {
  unsigned long long int i, j;
  double f, g;

#ifdef __i386__
  {
    unsigned short cw;
    asm("fnstcw %0" : "=m" (cw));
    cw &= ~0x0300u;
    cw |= 0x0200u;   /* double precision (64 bit) */
    asm("fldcw %0" : : "m" (cw));
  }
#endif

  i = 0x0008008000000000ull;   /* 1.11281e-308 */
  j = 0x3ff0000000000001ull;   /* 1.0 + alpha */

  printf("0x%016llx * 0x%016llx\n", i, j);   /* uses i & j */

  f = *((double *)&i);
  g = *((double *)&j);

  printf("%g (0x%016llx) * %g (0x%016llx) =\n", f, f, g, g);

  f = dmul(f, g);

  printf("%g (0x%016llx)\n", f, f);
}
```

SPARC など他のプロセッサでは、実行結果は次の通りです。

```
0x0008008000000000 * 0x3ff0000000000001
1.11281e-308 (0x0008008000000000) * 1 (0x3ff0000000000001) =
1.11281e-308 (0x0008008000000001)
```

しかし x86 ではこうなります。

```
0x0008008000000000 * 0x3ff0000000000001
1.11281e-308 (0x0008008000000000) * 1 (0x3ff0000000000001) =
1.11281e-308 (0x0008008000000000)
```

演算結果の末尾1ビットが、他のプロセッサでは1、x86では0となっている点に注意してください。

x86 の浮動小数点レジスタ

x86は、IEEE754で形式が規定されている単精度と倍精度に加えて、拡張倍精度という表現形式をサポートしています。それぞれの長さは32、64、80ビットです(**[Hack #97] 浮動小数点数のビット列表現**」参照)。浮動小数点演算器(FPU)が持つ制御レジスタの値を変更することで、仮数部の精度を設定できます。前掲したプログラムのインラインアセンブリコードは、fldcw 命令を実行することで精度の設定を行っていたのです。

ここから先がx86が独特である部分です。精度の設定が影響を与えるのは仮数部に対してだけであり、指数部はレジスタ上では常に 15 ビットなのです。単精度や倍精度に設定しても、8、11 ビットにはならず、常に 15 ビット分が保持されます(図 7-2)。

演算結果の絶対値が8、11ビットの指数部では表現できないくらい大きくなった場合、他のプロセッサでは結果は無限大となります。ところがx86の15ビット指数部ではこのあふれが起きない、ということが起きます。

次のプログラムを実行すると、SPARCなど他のプロセッサではあふれが起き、演算結果は無限大となります。ところが x86 ではあふれが起きません。

```
#include <stdio.h>

int main(int argc, char **argv) {
```

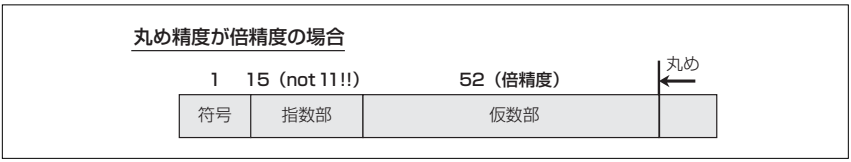


図 7-2 x86 プロセッサの特殊性

```
  unsigned long long int i, j, k;
  double f, g, h;

#ifdef __i386__
  {
    unsigned short cw;
    asm("fnstcw %0" : "=m" (cw));
    cw &= ~0x0300u;
    cw |= 0x0200u; /* double precision (64 bit) */
    asm("fldcw %0" : : "m" (cw));
  }
#endif

  i = 0x7fe0000000000000ull;   /* 1.0 x 2^1023 */
  k = j = i;
  printf("%016llx, %016llx, %016llx\n", i, j, k);   /* uses i, j and k */

  f = *((double *)&i);
  g = *((double *)&j);
  h = *((double *)&k);

  printf("%g + %g - %g = ", f, g, h);

  f += g;
  f -= h;

  printf("%g\n", f);
}
```

SPARCでの実行結果は次の通りです。結果は"Inf"、つまり無限大となっています。x86ではこのあふれが起きず、"Inf" の代わりに "8.98847e+307" となります。

```
7fe0000000000000, 7fe0000000000000, 7fe0000000000000
8.98847e+307 + 8.98847e+307 - 8.98847e+307 = Inf
```

対策

x86で他のプロセッサとまったく同一の演算結果を得ることは可能なのでしょうか。可能ではありますが、一筋縄では行きません。まず、1回演算を行うたびに結果をメモリに格納するという方法が考えられます。必要なら再びレジスタにロードするのです。指数部が常に15ビットであるのはレジスタ上の話であって、メモリに格納すれば精度に応じたビット数、単精度なら8、倍精度なら11ビットに丸められます。

しかし、この方法も完璧ではありません。演算時とメモリ格納時の2回、仮数部の丸めが起きて、(他のプロセッサで行われるように)1度で丸められた結果とは異なる値になってしまうことがあります。すなわち、アンダーフローが起きて、倍精度としては非正規数(denormalized number)として表されるべき値が、指数部が15ビットと余分にあるためにレ

ジスタ上では正規数(normalized number)として表現できてしまった場合です。この場合、メモリ格納時になって非正規数への変換が起き、ここで丸めが行われてしまいます。

冒頭で示した乗算のプログラム例は、まさにこの通り、丸めが2度起きてしまうという例です。この2度丸めを防ぐためには、メモリ格納時ではなく、演算時に適切にアンダーフローを起こす必要があります。これは、演算前にオペランドに対してある定数を乗じておき、演算後に定数の逆数を乗じることで、達成できます。

ここまでして初めて、x86でも他のプロセッサとまったく同一の演算結果を得ることができます。

SSE2命令

Pentium 4以降のx86プロセッサには、Streaming SIMD Extensions 2(SSE2)というSIMD演算命令セットがあります。SSE2はSIMD演算のための命令セットですが、オペランドを2つとる通常の浮動小数点演算もカバーしています。数値の表現形式はIEEE754の単精度と倍精度であり、四則演算、平方根、丸め方向の制御など、剰余を除いてIEEE754の要求をほぼサポートしています。

実は、SSE2以前の旧来のFP演算命令の代わりにSSE2命令を使うことで、他のプロセッサと同一の演算結果を得ることができます。SSE2では、指数部が常に15ビット、というようなことがないわけです。

Java言語のstrictfp

Java言語にはstrictfpという修飾子があります。これは、クラス、メソッド、インタフェースに付けることができ、その文脈では、x86以外のIEEE754準拠プロセッサとまったく同一の演算結果が得られるというものです。

つまり、x86上のJava仮想マシンはSSE2命令でFP演算を行っているか、もしくは、前述の複雑な対策を実行しているわけです。

まとめ

他のプロセッサとは異なる演算結果を導くことすらあるx86 FPUの特殊性を説明し、その回避手法を紹介しました。

—— Kazuyuki Shudo

HACK #99 結果が無限大や NaN になる演算でシグナルを発生させる

演算結果が無限大や非数値となった場合にシグナルを発生させる方法として、C99規格のヘッダ fenv.h を使う方法と x86 プロセッサ固有の方法を紹介します。

本Hackでは、浮動小数点演算で結果が無限大や非数値(NaN)となった際にシグナルを発生させる方法を紹介します。

IEEE754 の規定

浮動小数演算規格IEEE754(「**[Hack #97]　浮動小数点数のビット列表現**」参照)は、特に設定がないかぎりは、演算結果が無限大や非数値(NaN)になった場合でもプログラムの実行は中断させないよう、規定しています。このため、意図せずに無限大やNaNになってしまった場合でも、プログラムはその結果を使って処理を進めてしまいます。

ここでは、演算結果が無限大やNaNになった時点で検出する方法を述べます。glibcでの方法と、x86依存の方法を紹介します。

IEEE754 が定める例外と SIGFPE

演算結果が無限大やNaNとなった場合、現象に応じた例外が発生し、浮動小数点演算器(FPU)中の例外フラグがセットされます。

これらの例外がプロセッサの例外を引き起こすか否かは、FPUの設定次第です。プロセッサの例外は、UNIX系OSではSIGFPEというシグナルを発生させます。特に設定がないかぎり、プロセッサの例外、つまりシグナルは発生しません。このことはIEEE754自体が規定しています。

例えば、次のゼロ除算を実行すると演算結果は正の無限大となり、printf(3)で結果を表示させるとinfと表示されます。

```
double a = 1.0 / 0.0;
```

このままでは、意図せずに演算結果が無限大やNaNとなってしまった場合には困ったことになります。そのまま処理が進んでしまうので、どこで問題が起きたか、どのコードが原因なのかを特定することが難しくなりがちです。

そこで、以下では演算の時点でシグナルが発生させる方法を紹介します。シグナルが発生するならば、デバッガ経由でプログラムを実行させて、無限大やNaNの原因となった演算を特定できます。

glibc での方法

C99規格には、浮動小数点演算器(FPU)の丸めモードと例外を扱う型、マクロ、関数、プラグマを定義するfenv.hというヘッダがあります。fenv.hは、FPU中の例外フラグや、例外フラグの値(fexcept_t型)を扱う関数をいくつか宣言しています。しかしC99規格は、プロセッサの例外、つまりシグナルの発生を制御する方法までは提供していません。

glibcは、C99規格に対する独自の拡張として、シグナルの発生を制御する関数を提供しています。次のコードは、ある種の例外に対してシグナルが発生するようにFPUを設定します。1行目のようにマクロ _GNU_SOURCE を定義することで、glibc の独自拡張が有効になります。

```
#define _GNU_SOURCE
#include <fenv.h>

int excepts = fegetexcept();
excepts |= FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW;
feenableexcept(excepts);
```

これによって、さきほどのゼロ除算ではSIGFPEが発生するようになり、このシグナルをキャッチしないかぎりは、プログラムは次のように異常終了するようになります。

```
(実行)
Floating exception
```

指定できる例外の種類は次の通りです。

マクロ	条件
FE_INVALID	不法操作
FE_DIVBYZERO	ゼロ除算
FE_OVERFLOW	オーバーフロー
FE_UNDERFLOW	アンダーフロー
FE_INEXACT	精度落ち
FE_ALL_EXCEPT	サポートされている全例外の論理和

x86 依存の方法

x86では、FPU中の制御ワードレジスタを操作することで、浮動小数点演算の各種例外に対してプロセッサの例外を発生させることができます。特に設定しないかぎり、すべての例外はマスク(発生が抑制)されています。次のインラインアセンブリコードは一部のマスクを外します。

```
unsigned short cw;
asm("fnstcw %0" : "=m" (cw));
cw &= ~0xd;
asm("fldcw %0" : : "m" (cw));
```

ここでは、2行目で変数cwに制御ワードレジスタをロードし、3行目でマスクを外し、4行目でレジスタに値を戻しています。Linuxの場合、fpu_control.hヘッダを#includeして、次のように書くこともできます。

```
fpu_control_t cw;
_FPU_GETCW(cw);
cw &= ~(_FPU_MASK_IM | _FPU_MASK_ZM | _FPU_MASK_OM);
_FPU_SETCW(cw);
```

こういった設定によって、ゼロ除算などでSIGFPEが発生するようになります。この種の設定変更のために各OSには次のヘッダが用意されています。

```
Linux      fpu_control.h
FreeBSD    floatingpoint.h
Solaris    ieeefp.h
```

x86のFPU制御ワードレジスタには、マスクとして次の6ビットがあります。これらはそれぞれ例外の発生条件に対応しています。このうち、非正規数例外だけは、IEEE754では規定されていません。

マスク	値	条件
IM	0x1	不法操作
DM	0x2	非正規数
ZM	0x4	ゼロ除算
OM	0x8	オーバフロー
UM	0x10	アンダフロー
PM	0x20	精度落ち

前掲のプログラム中では、0xdに対応するマスクを外していました。つまり、IM、ZM、OMのビットをクリアしていたわけです。

まとめ

演算結果が無限大や非数値となった場合にシグナルを発生させる方法として、C99規格のヘッダfenv.hを使う方法とx86プロセッサ固有の方法を紹介しました。

—— Kazuyuki Shudo

HACK #100 文献案内

本Hackでは Binary Hack の参考になる文献を紹介します。

書籍

Write Great Code

「グレートコード」を書く上で必要となるコンピュータの基礎的な知識を幅広く扱った内容となっています。「ハードウェアを知り、ソフトウェアを書く」という副題の通り、『Write Great Code』では、CPU、キャッシュ、メモリ、ストレージ、周辺機器といったハードウェアの話題に多くの紙面が割かれています。本書を読みこなす上で必要になる基礎知識が多く含まれています。(Randall Hyde著、トップスタジオ訳、鵜飼文敏、後藤正徳、まつもとゆきひろ監訳、毎日コミュニケーションズ)

詳解 UNIX プログラミング

UNIXのシステムコール、ライブラリ関数ひとつひとつについて詳細に解説した貴重な本です。普段何気なく使っているシステムコールも、よくよくこの本の説明を読んでみると、さまざまな注意事項や意外な使い方などの発見があるはずです。歴史的背景についての記述も豊富であり、標準規格の解釈に厳密なのも特徴です。UNIXでシステムプログラミングをする上で必携の本です。(W・リチャード・スティーブンス著、大木敦雄訳、ピアソンエデュケーション)

コンピュータの構成と設計

著者の名前(パターソンとヘネシー)から「パタヘネ」という呼び名で知られる名著。コンピュータの基本概念から、命令セット、算術演算、性能評価、プロセッサやコンピュータシステムのアーキテクチャを解説しており、ハードウェアとソフトウェアとの相互関係を理解することができます。ソフトウェア側からだけでなく、ハードウェア側からみてどのように実行されるかを理解するために必要な知識を得ることができます。(上／下巻、デイビッド・A・パターソン、ジョン・L・ヘネシー著、成田光彰訳、日経BP社)

Linkers&Loaders

リンカとローダについての数少ない専門書です。リンカとローダの仕組みだけでなく、システムプログラミングに役立つ知識も得られる本です。前半では、リンカに関係するOSの仕組みについて、たとえば仮想メモリとファイルのマッピングの関係や、マップ時の書き込み時コピーの仕組み、といった内容が歴史的経緯とともに説明されています。後半ではリンカの仕組みの詳細に入り、位置独立コードの実現手法や、動的リンクの仕組みが取り上げら

れています。本書にたびたび登場するリンカ、ローダの話題について詳しく知りたい方にお勧めの本です。(John R. Levine 著、榊原一矢監訳、ポジティブエッジ訳、オーム社)

デバッガの理論と実装

その名の通り、デバッガの理論と実装についての解説書です。特定のデバッガに限定した内容ではなく、各種 OS のデバッガや、Java のデバッガなど、さまざまなデバッガに関する話題を幅広く扱っています。デバッガの実装は実行環境の OS やプロセッサに密接に関わっていますが、ブレークポイントやステッピングなどの基本的なコンセプトはどのデバッガにも共通しています。本書では、これらのデバッグ機能を実現するために提供されている OS の API とプロセッサの機能について詳しく解説されています。(Jonathan B. Rosenberg 著、吉川邦夫訳、アスキー)

ハッカーのたのしみ

バイナリといえば二進、二進といえばビットです。ビット単位の操作を含め、細かく繊細な、さまざまな操作について述べている良書です。(ヘンリー・S・ウォーレン、ジュニア著、滝沢徹、鈴木貢、赤池英夫、葛毅、藤波順久、玉井浩訳、エスアイビー・アクセス)

セキュリティウォリア

情報セキュリティ技術一般を扱っている書籍です。3章の「Linux リバースエンジニアリング」には、本書を読む上で参考になる記述があります。たとえば、nm、gdb、lsof、ltrace、objdump などの基本的なツールの使い方の解説、objdump による逆アセンブル結果の読み方の解説、ptrace(2)を利用した簡単なツールの開発例、GNU BFD(特に libopcodes.a)の活用例などが書かれています、原書、日本語版ともに、表紙は力士のイラストです。(Cyrus Peikari、Anton Chuvakin著、西原啓輔監訳、伊藤真浩、岸信之、進藤成純訳、オライリー・ジャパン)

JIS X 3010:2003 プログラム言語C

最新のC言語規格である ISO/IEC 9899:1999(通称:C99)を、日本語に翻訳したものです。C コンパイラを Hack する際には必携の書といえます。この規格書には、他ではなかなか知ることのできない微妙な部分も含め、C 言語のすべてが記載されています。たとえば、C 言語における整数および浮動小数点数の細かな演算規則はWeb上にはまとまった解説が見当たりませんが、この規格を読めば完全に理解することができます。本書の「**[Hack #43] -ftrapv で整数演算のオーバーフローを検出する**」、「**[Hack #47] bitmask する定数は符号なしにする**」、「**[Hack #99] 結果が無限大や NaN になる演算でシグナルを発生させる**」といった Hack のより深い理解にも役立つことでしょう。財団法人日本規格協会の WebStore(http://www.webstore.jsa.or.jp/)より誰でも購入可能です。

インターネット

How To Write Shared Libraries

Linuxの共有ライブラリ(動的共有オブジェクト)に関する決定版的なドキュメントです。LinuxにELFバイナリが導入された歴史的経緯に始まり、共有ライブラリのメリット、デメリット、動的リンクの仕組み、性能の改善方法など、さまざまな話題が凝縮されています。共有ライブラリを用いたHackを行う際に必要な情報が網羅されているといっていいでしょう。著者のUlrich Drepper氏のサイト(http://people.redhat.com/~drepper/)にはこの他にもLinuxのスレッドライブラリやSELinuxなどに関する貴重な情報が多数掲載されています。バイナリアンまっしぐらのサイトです。(http://people.redhat.com/~drepper/dsohowto.pdf)

The Single UNIX Specification, Version 3

UNIXの規格です。次の4つの部分(volume) から構成されています。

- Base Definitions (XBD):用語定義、ファイル、ファイルパーミッションなどのUNIXの基本コンセプトの解説、さらにC言語向けヘッダファイルの定義など。
- System Interfaces (XSH):シグナル、ソケット、スレッドなどのインタフェースに関する全体的な説明と、個々の関数、システムコールの定義など。
- Shell and Utilities (XCU):ls、catなどの個々のコマンドの定義、シェルスクリプトの文法定義など。
- Rationale (XRAT):上記3つの規格にうまく収まらない部分や、規格制定までの経緯の説明など。

The Single UNIX Specificationは、UNIXの深い学習に有用なドキュメントです。特にXSHのAPI一覧は、眺めていると思わぬ発見があるものです。一部のLinuxディストリビューションには、manとしてXSHとXCUの内容が収録されています。セクション0p(ヘッダ)、1p(コマンド)、3p(関数&システムコール)のmanを探してみてください。またmozdev.orgでは、この規格をFirefoxの検索バーから検索するためのプラグインが配布されています。mozdev.orgのホームページから"Single UNIX Specification"で検索してください。(http://www.unix.org/single_unix_specification/)

arbitrary unix stuff

各種UNIXの#!の実装や各種シェルの内蔵echoコマンドの動作など、UNIXに関するトリビア的な情報が満載のサイトです。UNIXのさまざまな機能のディテールに興味のある人向けのサイトです。(http://www.in-ulm.de/~mascheck/various/)

GCC のマニュアル

GCCの起動オプションや、C/C++の拡張機能などについて書かれたドキュメントです。起動オプションや、__attribute__ 拡張の部分は一度目を通しておいて損はないかと思います。インラインアセンブラのマニュアルもここにあります。

- gcc-2.95.3に付属のTexinfo形式マニュアルの日本語訳（http://www.sra.co.jp/wingnut/gcc/）
- 英語の最新版（http://gcc.gnu.org/onlinedocs/gcc/）
- コマンドラインオプション関連
  GCC Command Options（http://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html）
- __attribute__ 関連
  Attribute Syntax（http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html）
  C++ Attributes（http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html）
  Type Attributes（http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html）
  Function Attributes（http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html）
  Variable Attributes（http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html）
- インラインアセンブラ関連
  Assembler Instructions with C Expression Operands（http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html）
  Constraints for asm Operands（http://gcc.gnu.org/onlinedocs/gcc/Constraints.html）
- その他
  Pragmas Accepted by GCC（http://gcc.gnu.org/onlinedocs/gcc/Pragmas.html）

GNU ツールチェインの info

本書でたびたび登場した、定番ソフトウェアのマニュアルです。Webにあるマニュアルが最新とはかぎらないので注意しましょう。

- info libc（http://www.gnu.org/software/libc/manual/html_node/index.html）
- info binutils（http://www.gnu.org/software/binutils/manual/html_node/binutils_toc.html）
- info gdb（http://www.gnu.org/software/gdb/documentation/）
- info bfd（http://www.gnu.org/software/binutils/manual/bfd-2.9.1/html_node/bfd_toc.html）

GCC Wiki

GCCに関する情報が集まっているWikiです。GCCの開発に関する情報や、GCC最新機能に関する話題が多く載っています。（http://gcc.gnu.org/wiki/）

comp.lang.c Frequently Asked Questions

Usenetのcomp.lang.cで作られたC言語に関するFAQです。"C FAQ"として知られています。初、中級者がC言語で陥りやすいポイントを適確に指摘しています。書籍にもなっています。(http://www.c-faq.com/)

マイクロプロセッサアーキテクチャマニュアル

IA-32(x86)、EM64T、IA-64(IPF)

- 「インテル 日本語技術資料のダウンロードページ」(http://www.intel.com/jp/developer/download/index.htm)。
  Intelのマニュアル、アプリケーションノート、データシートの日本語翻訳が集められているページです。ソフト開発を行う時に欠かすことのできない「IA-32 Intel Architecture Software Developer's Manuals」、「Itanium Architecture Software Developer's Manauls」、最適化手法の解説の日本語訳がダウンロード可能です。最新の英語版ドキュメントへのリンクもあり便利。

AMD x86、AMD64

- 「AMD Developer Central - Documentation」(http://developer.amd.com/documentation.aspx)。
  本家AMD64の命令セット仕様書である「AMD64 Architecture Programmer's Manual」やAthlon64、Opteron向けの最適化ガイドがあります。

Alpha

- 「Alpha technical documentation library」(http://h18002.www1.hp.com/alphaserver/technology/chip-docs.html)
  Alpha 21x64シリーズの命令セットを解説した「Alpha Architecture Handbook」、「Compiler Writer's Guide for the Alpha 21264」や個々のチップの仕様がダウンロード可能です。
- 「Tru64 UNIX Version 4.0F Online Documentation」(http://h30097.www3.hp.com/docs/pub_page/V40F_DOCS.HTM)。「Digital UNIX Assembly Language Programmer's Guide」というAlphaのABIを解説した重要な資料があります。

ARM

- 「ARM：マニュアルダウンロード」(http://www.jp.arm.com/document/manual/)
  各種日本語マニュアルがダウンロード可能なページ(要無料登録)です。命令セットの

解説は「RealView コード生成ツール v2.0　アセンブラガイド」に含まれています。

MIPS

- 「MIPS IV Instruction Set」（http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=hdwr&db=bks&docnumber=007-2597-001）
  命令セットを解説しています。
- 「MIPSpro N32 ABI Handbook」（http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=0650&db=bks&docnumber=007-2816-005）
  IRIX の ABI の解説です。
- 「MIPSpro Assembly Language Programmer's Guide」（http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=0650&db=bks&docnumber=007-2418-006）
  アセンブラによるプログラミング方法の解説です。
- 「MIPS Technologies, Inc」（http://www.mips.com/、日本語：http://www.mips.jp/）
  MIPS32、MIPS64 の仕様です。

PA-RISC

- 「PA-RISC 1.1 architecture and instruction set reference manual」（http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,958,00.html）
- 「PA-RISC 2.0 Architecture」（http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,2533,00.html）

PowerPC

- 「IBM Microelectronics - PowerPC」（http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC）
  命令セット仕様書、IBMのPowerPCプロセッサのユーザーマニュアルがダウンロード可能です。
- 「Freescale Semiconductor - PowerPC Processors」（http://www.freescale.com/powerpc）
  Apple 社の Macintosh に PowerPC を提供していた旧モトローラの半導体部門が分離独立した会社のページです。Documentationの項から各プロセッサのリファレンスマニュアルなどがダウンロードできます。

SH

- 「ルネサステクノロジ ドキュメント」（http://japan.renesas.com/fmwk.jsp?cnt=Documentation.jsp&fp=/products/mpumcu/superh_family/&title=%E3%83%89%E3%82%AD%E3%83%A5%E3%83%A1%E3%83%B3%E3%83%88&lid=2# ソフトウェアマニュアル）

命令セットの解説など、各種日本語マニュアルがダウンロード可能です。

SPARC

- 「SPARC International, Inc. - Standards」（http://www.sparc.org/standards.html）
  SPARC 標準団体のページです。命令セット（「The SPARC Architecture Manual V8 & V9」）や ABI 仕様書がダウンロード可能です。
- 「UltraSPARC Processors Document」（http://www.sun.com/processors/documentation.html）
  Sun Microsystems の UltraSPARC プロセッサのユーザーガイドがダウンロード可能です。
- 「OpenSPARC.net」（http://opensparc.sunsource.net）
  UltraSPARC T1 チップのプロセッサコードと仕様がオープンソース化された OpenSPARC Projectのページです。UltraSPARCのアーキテクチャ仕様書や仮想化機能の仕様書をダウンロードすることができます。

S/370

- 「ESA/390 Principles of Operation」（http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9AR006/CCONTENTS）
- 「z/Architecture Principles of Operation」（http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9ZR000/CCONTENTS）
  「Principles of Operation」（PoO）は、IBM System/360（S/360のアーキテクチャマニュアルです。製品マニュアルですが、アーキテクチャの教科書としても広く使われてきました。S/360 がS/370、ESA/390、ZSeries と進化するのに対応し PoO も改訂されています。

索　引

記号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

あ行

か行

さ行

た行

な行

は行

ま行

や行

ら行

Binary Hacks
――ハッカー秘伝のテクニック100選

2006年11月8日　初版第1刷発行
2013年1月31日　初版第14刷発行

著　　者　高林 哲(たかばやし さとる)、鵜飼 文敏(うかい ふみとし)、
佐藤 祐介(さとう ゆうすけ)、浜地 慎一郎(はまじ しんいちろう)、
首藤 一幸(しゅどう かずゆき)
発 行 人　ティム・オライリー
印刷・製本　株式会社平河工業社
発 行 所　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
Tel　(03)3356-5227
Fax　(03)3356-5263
電子メール　japan@oreilly.co.jp
発 売 元　株式会社オーム社
〒101-8460　東京都千代田区神田錦町3-1
Tel　(03)3233-0641(代表)
Fax　(03)3233-3440

Printed in Japan(ISBN4-87311-288-5)
乱丁、落丁の際はお取り替えいたします。