論　文

動的リンカを用いた組込み Linux のセキュリティ向上技術の開発

大原　衛[*1)]　　岡野　宏[*2)]

A software-based approach for improving security of embedded Linux systems using dynamic linkers

Mamoru Ohara[*1)], Hiroshi Okano[*2)]

Growing number of embedded systems are connected to the Internet recently. For example, we can read E-mails and browse home pages with the digital TVs. Today, we have to consider the network security of such networked embedded systems. Traditional security techniques used in PCs are often too complicated to apply them to the embedded systems because the embedded systems are usually poor in hardware resources. We also have difficulties in developing general-purpose secure hardware for the embedded systems due to the wide variety of their hardware constructions. In this paper, we propose a software approach to improve network security of the embedded systems. We made alternations to a Linux dynamic linker, which is a part of the programming language processors, so that the linker dynamically modifies codes in insecure programs just before running them. We implemented the dynamic linker for a testbed system consisting of the ARM processor and embedded Linux and examined the effectiveness of the proposed technique by running some programs having buffer-overflow vulnerabilities. We could confirm that the dynamic linker could detect the attacks to the vulnerabilities and gracefully handle them.

キーワード：組込みシステム，組込み Linux, セキュリティ，スタック破壊脆弱性, 動的リンカ
Keywords：Embedded systems, embedded Linux, security, stack-smashing attacks, dynamic linkers

1. まえがき

近年，コンピュータネットワークに接続される組込み機器が増加し，ネットワークの利用目的も多様化している。このような組込み機器において，ネットワークセキュリティの確保が急務となっている。これまで数多くのセキュリティ対策手法が提案されてきたが，これらの多くは PC などの比較的高性能な計算機を対象としたものであり，ハードウェアリソースの乏しい組込み機器には適用が困難な場合がある。このため，組込み機器に適用できるセキュリティ対策技術が求められている。

本研究では，組込み機器のネットワークセキュリティを向上させるソフトウェア技術の研究開発を行った。組込み機器は比較的ハードウェアリソースが乏しく，多様性が高いため，特定のハードウェア機能に依存した技術は，十分な汎用性を確保できない場合がある。ソフトウェアによる対策技術は，組込み機器のハードウェアの多様性を隠蔽し，多くの組込み機器に広く適用できる可能性がある。

本稿では，スタック破壊攻撃を検出する機能を，ユーザプログラムに動的に付加する手法を提案する。スタック破壊攻撃は，最も頻繁に利用される攻撃手法の一つである。提案手法は，動的リンカと呼ばれる特別なプログラムを変更して，ユーザプログラムを実行時に書き換えられるようにする。動的リンカなどの実行時処理系を改変する手法は，他の手法に比べて，カーネルを含めた既存ソフトウェアに加える必要のある修正が少ない。このため，提案手法を用いることで，既存ソフトウェア資産にほとんど修正を加えることなく，セキュリティの向上を図ることができる。

本研究では，ARM プロセッサ上で動作する Linux 向けの動的リンカを開発した。近年，組込み機器は多機能化を求められており，Linux のような汎用 OS が採用される製品例が増加している。開発した Linux 向け動的リンカを用いて，スタック破壊攻撃を検知・対応できることを確認した。

本稿は以下のように構成される。2 節では，これまでに提案された関連する研究について概説する。3 節は，本稿で議論するスタック破壊攻撃の原理について述べる。4 節は本研究で行なった技術開発について報告する。5 節は本稿のまとめと今後の課題を示す。

2. 関連研究

コンピュータシステムの脆弱性を攻撃する手法は，通常以下の 2 段階の手順で攻撃を行なう。すなわち，

① システムに不正なプログラムコードを挿入する。
② 挿入したプログラムコードを実行させる。

プログラムに与えられた入力がデータであるかプログラムであるかを判別するのは困難であるため，①は本質的に対策が難しい。このため，これまでに提案されているセキュリティ技術には，②の攻撃手順の無効化を図るものが多い。

[*1)] IT グループ
[*2)] エレクトロニクスグループ

①の手順は，例えばシステムのメモリを随時監視して，既知の不正なプログラムのビットパターンと合致するコードをメモリから削除するなどの手法で防ぐことができる。このような手法は，PC 向けウィルス対策ソフトウェア等で用いられているが，多量のビットパターンを蓄積しなければならないため，組込み機器への適用が難しい。

②の手順に対する対策技術は，ソフトウェアを用いるもの[(1)(2)]とハードウェアに機能を付加するもの[(3)(4)(5)]がそれぞれ提案されている。ソフトウェアを用いる手法は，静的な手法と動的な手法に大別できる。静的な手法の代表的なものは，コンパイラに機能を付加する手法である[(1)]。この手法は，コンパイラがプログラムのソースコードの情報を得られるため，プログラム作成者の意図を反映しやすい点で優れている。静的な手法の欠点は，既にコンパイル済みのプログラムに対して適用できないことである。商用プログラムのソースコードの入手は困難であることが多いため，これらの手法は適用できない場合がある。

動的な手法は，プログラムの実行時処理系を変更する。この手法の代表的な例として，Baratloo らによる libsafe と libverify が挙げられる[(2)]。libsafe は，C 言語の標準関数のうち，使い方を誤るとプログラムに脆弱性を与える可能性のあるものを，特別なライブラリによって置き換える。プログラム実行時に使用するライブラリを変更することは容易であるため，すでに複数の製品で採用されている。しかし，C 言語の標準関数以外には適用できない。

また，libverify は，プログラム開始直前にプログラム中のすべての関数をヒープ領域にコピーし，このコピーに後述するような戻りアドレスの保護機能を付加する。さらにプログラム中のすべての関数呼び出しをこのコピーを呼び出すよう書き換えることで，既存ソフトウェアにセキュリティ機能を追加することができる。プログラム開始直前に動的にコードを書き換えて戻りアドレスを保護する点で，libverify は提案手法と類似している。

本稿の提案手法は，このような既存の動的な手法と相補的に用いられることを想定している。提案手法は，ライブラリ化されていないユーザ関数の脆弱性を修正するが，ライブラリ関数は修正しない。このため，既に libsafe などの対策技術を導入しているシステムにも提案手法を適用することが可能である。

ハードウェアに機能を付加する手法は，実行時の性能オーバヘッドをほとんど伴わない点で優れている。Intel 社や AMD 社のプロセッサは，メモリ管理ユニット（MMU）のページテーブルを拡張し，メモリページごとに実行の可否を制御することができる[(3)(4)]。実行が禁止されたメモリページ上にプログラムの制御が移ると，MMU がこれを検知して割り込みを生じさせる。

Lee らは，近年のプロセッサが持つリターンアドレススタック（Return Address Stack: RAS）を利用した SRAS（Secure RAS）手法を提案した[(5)]。RAS は，プロセッサの投機実行の効率を向上する目的で導入され，関数の呼出履歴を保持している。これを用いて，プログラムの実行フローの変更を検知する。これらのような追加的なセキュリティ機能を持ったハードウェアは比較的高価であるため，組込み製品での採用実績は少ない。

3. スタック破壊攻撃

スタック破壊攻撃は，バッファオーバフローと呼ばれるソフトウェア脆弱性を利用する攻撃手法である。この脆弱性は，主に C 言語などで開発されたプログラムに多く見られる。図１は，C 言語などで開発されたプログラムにおけるスタック領域の利用例を示している。この例では，プログラムに関数 A と B が存在し，関数 A がその処理の途中で関数 B を呼び出すとする。また，関数 A は a，b 二つの変数を使用し，関数 B は 16 文字以内の入力を外部から受け取って変数 input に格納するものとする。

バッファオーバフロー脆弱性を持つプログラムは，想定を超えた量のデータを入力された場合に対応することができない。この例では，16 文字を超えるデータを入力されると，変数 input にデータが収まりきらず，その直下の戻りアドレスを書き換えられてしまう可能性がある。攻撃者は，このように戻りアドレスを書き換え，プログラムの動作を任意に変更することができる。

4. 提案手法

本稿の提案手法は，プログラムの実行直前に動的にコードを書き換えて戻りアドレスの保護機能を追加する。関数の呼出直前と復帰直前に追加的なコードを挿入し，以下の処理を行うことで，戻りアドレスが不正に書き換えられていないことを検証する。

① 関数が呼び出され，その処理が行われる直前に，戻りアドレスを含むスタック領域の内容を別のメモリ領域にコピーする。

② 関数が復帰する直前に，前項で作成したコピーとスタック領域の内容を比較し，戻りアドレスの書き換えの有無を検証する。

図２は，提案手法の追加する①および②のセキュリティ機能とプログラム中の任意の関数 A および，A から呼び出

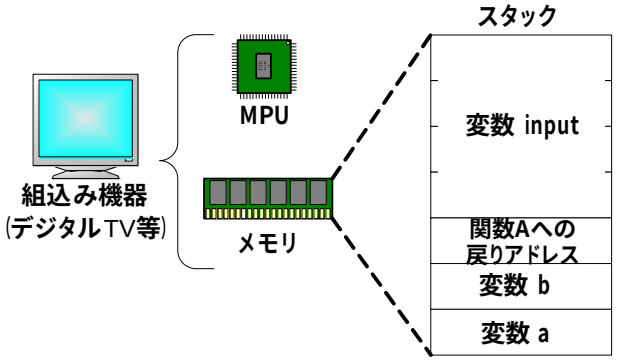

図１．組込み Linux におけるスタック領域の利用例

される関数 B の関係を模式的に表したものである。通常のプログラムでは，関数の呼出は，図の関数 A に示したような 2 段階の手順で行なわれる。すなわち，関数に与える引数をスタック領域にコピーし，コール命令を実行して目的関数に制御を移す。呼び出された関数 B は，まず戻りアドレスをスタックに保存し，関数の機能を実現するためのいくつかの命令を実行した後，リターン命令を実行する。

提案手法のプログラム処理系は，プログラムの実行前に呼出コード片と復帰コード片を自動的に生成する。次に，プログラム中のコール命令のオペランドを書き換え，呼び出し先を呼出コード片に変更する。また，リターン命令を復帰コード片へのジャンプ命令に書き換え，スタックの内容に係らず必ず復帰コード片が実行されるようにする。呼出コード片は，スタック領域のコピーを行った後，本来のコール命令を実行する。復帰コード片は，スタック領域中の戻りアドレスの改変の有無を検証し，改変がなかった場合はリターン命令を実行する。改変が検出された場合は，ソフトウェア例外を発生させ，プログラムに通知する。

4. 1　Linux のプログラム実行手順と動的リンカ　Linux における ELF（Executable and Linking Format）形式プログラムの実行開始フローを図 3 に示す。ユーザがプログラム a の実行を指示すると，カーネルは動的リンカと呼ばれる特別なプログラムを実行する。

動的リンカは，カーネルから実行すべきプログラム名 a を受け取り，このファイルの冒頭に置かれたヘッダを読み込む。動的リンカは，このヘッダに従ってプログラム a および必要な共有ライブラリをメモリに読み込む。

次に，動的リンカはプログラムの再配置処理を行う。再配置処理では，プログラム a 内で共有ライブラリ内の関数を呼び出している命令を検索し，命令オペランドの呼び出し先アドレスを共有ライブラリが実際に読み込まれたアドレスを基に修正する。最後に，動的リンカはカーネルに処理を戻し，カーネルはプログラム a が実行可能になったと判断して，a をスケジュール可能にする。

このように，Linux において，動的リンカはプログラムの読み込みとメモリ内への配置を担当する。本研究では，動的リンカに改造を加え，再配置を完了しカーネルに処理を戻す前に，上述したスタックのコピーおよび検証を行なうコードを生成する手順を追加する。

4. 2　PLT と GOT　今日の多くの OS では，ライブラリは複数のプログラム間で共有される共有ライブラリとして提供される。共有ライブラリの作成時には，このライブラリがどのようなプログラムから利用されるのかを想定することはできない。このため，実行時にプログラムと共有ライブラリの調整を行なう目的で，前述の動的リンカによる再配置処理が行われる。ELF 形式のプログラムでは，再配置は Procedure Linkage Table（PLT）と Global Offset Table（GOT）と呼ばれる仕組みを用いて実現される。

提案手法では，libsafe などの既存のセキュリティ対策手法

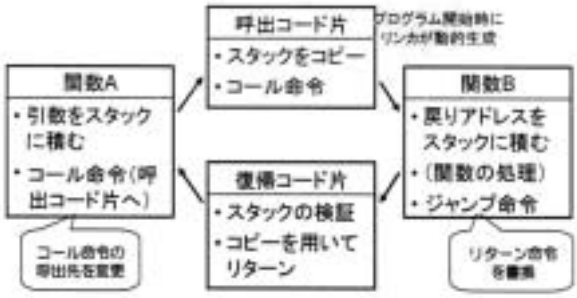

図 2．提案手法における関数の呼出・復帰フロー

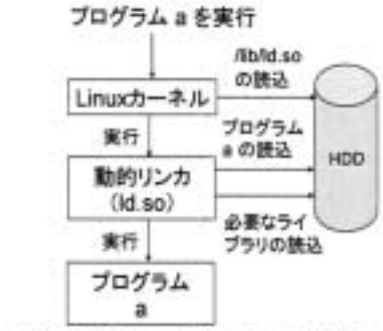

図 3．組込み Linux におけるプログラムの実行開始フロー

の併用を可能にするため，共有ライブラリ中の関数については戻りアドレスの保護機能を追加しない。提案手法の動的リンカは，プログラムと共有ライブラリ間の再配置を完了したあと，後述の手順でプログラムを走査し，関数呼び出しと復帰に関するコードを書き換える。この際に，あらかじめプログラムの PLT および GOT を調べることで，共有ライブラリに含まれる関数を判別し，これらに関する書き換え処理を省略する。

4. 3　提案手法の実装　提案手法を図 4 に示したマイコンボード上に試験的に実装し，改造した動的リンカの動作を確認した。マイコンボードは，Linux と組み合わせて組み込まれることの多い英 ARM 社のプロセッサを搭載する。このマイコンボード上に，Linux 2.6 カーネルと C 言語の実行時処理系である GNU glibc 2.5 を組み合わせて実装した。

通常，ARM アーキテクチャ上の C コンパイラは，関数呼び出しを BL 機械語命令に変換する。BL 命令は，リンクポインタと呼ばれるレジスタに戻りアドレスを格納して，指定されたアドレスにジャンプする命令である。関数呼び出しが 2 重以上の入れ子になる場合に，リンクポインタに格納された戻りアドレスをスタックに退避するのは，呼び出された関数の役割である。試験実装では，以下の処理を行って関数呼び出し時にスタックのコピーを行なわせるコードを挿入した。①メモリに読み込まれたプログラムを線形探索して BL 命令を発見する。②BL 命令のオペランドには呼び出される関数の相対アドレスが指定されている。この関数の絶対アドレスを計算し，これが初出のものであれば対応する呼出コード片を生成する。ただし，絶対アドレス

が PLT を指していれば，この BL 命令を無視して手順①に戻る。③BL 命令のオペランドを，対応する呼出コード片の先頭アドレスに書き換える。フローチャートを図５に示す。

ARM アーキテクチャでは，関数からの復帰を表現できる機械語命令列が数種類存在する。試験実装では，簡単のため，GNU gcc 3.4 コンパイラが生成する機械語を参考にして，最も使用される頻度の高い LDMDB 命令を対象に書き換えを行なった。LDMDB 命令は，オペランドで指定されたアドレスから連続する複数語を，複数のレジスタにロードする。これを用いて，スタックからの関数コンテキストの復帰を 1 命令で実行できる。コードの書き換えには，前述の BL 命令の書き換えと同様のアルゴリズムを用いた。

以上のアルゴリズムを実行する動的リンカを作成し，これを用いた処理系で，小規模なプログラム prime を実行した。prime は，引数として与えられた数以下の素数を発見する自作のプログラムである。まず，prime が正常に実行できることを確認した。

次に，prime に意図的にバッファオーバフロー脆弱性を作りこみ，スタック破壊実験を行なった。prime は 1000 個の要素を持つ配列に発見した素数を格納することとし，1000 個以上の素数が発見された場合には戻りアドレスが書き換えられるようにした。この実験の結果，引数として 10000 を指定した場合に，復帰コード片がスタックの破壊を検知し，ソフトウェア例外を発生させることを確認した。

提案手法の時間オーバヘッドは，プログラムの構造に大きく依存する。頻繁に関数を呼び出すプログラムほど性能の劣化が大きい。prime は，3 重ループの最も内側で非常に頻繁に関数を呼び出すため，提案手法の処理系では，通常のリンカを用いた場合の 8 倍程度の処理時間を要した。一方，ループ内の関数呼び出し回数を最小化するようにプログラムを変更した場合，提案手法はほとんど時間オーバヘッドを伴わなかった。

提案手法の空間オーバヘッドは 2 つの要因で増加する。まず，動的リンカが呼出・復帰コード片を追加するために，１つの呼出・復帰の組について 20 語程度のメモリを静的に消費する。また，戻りアドレスのコピーのために動的に消費するメモリ量は，1 回の関数呼び出しについて 2 語である。動的に消費されるメモリは，関数の復帰後には再利用することができるので，平均的には許容できるオーバヘッドであると考えられる。静的な空間オーバヘッドの大きさは，プログラムに含まれる関数コール，リターン命令の個数に依存する。例えば，prime では 20 か所の命令を書き換え，約 800 バイトのメモリを消費する。

一方，busybox のようなプログラムの書き換えには多量（数百 k バイト）のメモリを必要とした。busybox は，ls や cat 等の Linux の基本的なコマンドを，ディスク容量を節約する目的で複数集成したプログラムである。このため，通常，多数の関数を含む。busybox は組込み Linux で頻繁に用いられるため，このようなプログラムに対する空間オーバヘッドの削減は今後の主要な課題である。

図４．試験実装機

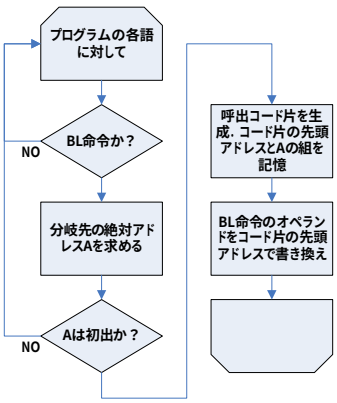

図５．BL 命令の書き換えアルゴリズム

5. まとめ

本研究では，組込み Linux の動的リンカを改造して，プログラムを実行直前に動的に書き換えることで，スタック破壊攻撃の有無を検出可能にする技術を開発した。制作した動的リンカを用いて動作実験を行い，バッファオーバフロー脆弱性を有するプログラムに不正な入力を行なった際に，提案手法によってこれを検出できることを確認した。

提案手法は，プログラムを動的に修正するため，既存のソフトウェア資産を再コンパイルすることなくセキュリティの向上を図ることができる。また，libsafe 等の従来手法を相補的に用いてセキュリティの向上を図ることができる。

今後の課題として，busybox などのプログラムの書き換えにおける空間オーバヘッドの削減が挙げられる。

（平成 19 年 6 月 29 日受付，平成 19 年 8 月 22 日再受付）

文　献

(1)C. Cripson et. al: “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” Proc. 7th USENIX Security Sympo., pp. 63-78 (1998)

(2)A. Baratloo et. al: “Transparent Run-Time Defence against Stack Smashing Attacks,” Proc. USENIX Annual Technical Conf. (2000)

(3)S. Kuo: Executable Disable Bit Functionality Blocks Malware Code Execution, available at http://www.intel.com/cd/channel/reseller/ijkk/jpn/products/250637.htm

(4)Microsoft TechNet, Changes to Functionality in Microsoft Windows XP Service Pack 2 – Part 3: Memory Protection Technologies, available at http://technet.microsoft.com/en-us/library/bb457155.aspx

(5)R. Lee et. al: “Enlisting Hardware Architecture to Thwart Malicious Code Injection,” Proc. Int’l Conf. Security in Pervasive Computing, pp. 237-252 (2003)