Vol.2015-DPS-162 No.33
Vol.2015-CSEC-68 No.33
2015/3/5

QEMUを用いた命令拡張による リターンアドレス書換え攻撃検知手法

柴田 達也[1,a)]　奥野 航平[1]　大月 勇人[1]　瀧本 栄二[1]　毛利 公一[1]

概要：サイバー攻撃が大きな問題となっている．サイバー攻撃では，ソフトウェアの脆弱性が悪用されることが多い．特に，バッファオーバーフロー脆弱性を悪用するものは従来から多く存在する．この攻撃は，リターンアドレスを書き換えることで任意の処理を行う．このような書換えを排除できれば対策が可能であるが，従来手法は緩和策にとどまっており，決定的な解決法とはなっていない．以上の背景から，本論文では，リターンアドレスを書き換えられた場合に，それを検出可能とするセキュアプロセッサの実現を目指す．具体的には，プロセッサにリターンアドレスの書換えを検出するための機能と命令を追加する方式を提案する．また，提案方式の有効性を検証するために，ハードウェアエミュレータに拡張機能を実装したのでそれについて述べる．さらに，当該機能を利用するOSのプロトタイプ実装についても述べる．また，このOSでテストプログラムを用いたテストを行ったのでこれについても述べる．

キーワード：バッファオーバーフロー攻撃，Mimicry Attack，シャドウスタック，QEMU

A Detection Method of Return Address Overwriting Attacks Based on Instruction Extension Using QEMU

Tatsuya Shibata[1,a)]　Kohei Okuno[1]　Yuto Otsuki[1]　Eiji Takimoto[1]　Koichi Mouri[1]

Abstract: Cyber-attacks are major threat. Commonly, these attacks attempt to exploit software vulnerabilities. In particular, attackers have exploited buffer overflow vulnerabilities for a long time. Many protection techniques were proposed to protect from buffer overflow attacks, but these techniques cannot prevent attacks completely. In this paper, we present a return address overwriting detection system. Our system disallows to attackers to exploit buffer overflow vulnerabilities. Our system works on CPU which has an extended feature and additional instructions. We implemented it to the hardware emulator QEMU. Furthermore, we confirmed the system with test programs on a prototype OS.

Keywords: Buffer Overflow Attack, Mimicry Attack, Shadow Stack, QEMU

1. はじめに

サイバー攻撃が大きな問題となっている．サイバー攻撃では，ソフトウェアの脆弱性が悪用されることが多い．特に，バッファオーバーフロー脆弱性を悪用する攻撃は従来から多く存在し，個人情報の流出やシステム破壊などの要因となっている．バッファオーバーフロー脆弱性を悪用した攻撃では，攻撃者は関数の戻り先を示すリターンアドレスを書き換え，制御フローを開発者の意図しないものに変更して任意の処理を行う．ソフトウェアの脆弱性を事前に全て見つけ出すことは困難であることから，やはり攻撃を完全に防ぐことも困難である．既存の主なバッファオーバーフロー攻撃への対策として，スタック保護 [1]，データ実行防止・アドレス空間のランダマイズ [2], [3]，ホスト型侵入検知システム [4], [5] が存在する．しかし，これらの対策は一定の効果はあるものの緩和策にとどまっており，決

[1] 立命館大学
525-8577 滋賀県草津市野路東 1-1-1
[a)] tsibata@asl.cs.ritsumei.ac.jp

© 2015 Information Processing Society of Japan

定的な解決策とはなっていない．例えば ROP 攻撃 [6] や mimicry attack [7]，[8] は，これらの対策を回避しながら攻撃を行う．本論文では，ROP 攻撃や mimicry attack など，リターンアドレスを書き換えて制御を奪う攻撃を総称してリターンアドレス書換え攻撃と呼ぶ．

リターンアドレス書換え攻撃を完全に防止する先行研究として，スタック領域のリターンアドレス検査によるスタック偽装攻撃検知手法 [9] が提案されている．この手法は，関数の入口でリターンアドレスの保存を行い，関数の出口で保存した全てのリターンアドレスを比較することでリターンアドレス書換え攻撃，特に mimicry attack を検知する．このように，リターンアドレスの検査を行うことで，リターンアドレス書換え攻撃を完全に防ぐことが可能となる．しかし，この手法を実現するためには，ソフトウェアのコンパイル時にリターンアドレスの保存と比較を行うためのコードをソフトウェアに追加する必要がある．そのため，この手法はオープンソースソフトウェアにのみ適用可能であり，プロプライエタリなソフトウェアに対して適用することはできない．

そこで，本論文では，ハードウェアの機能を拡張し，その機能を OS が利用することでプロプライエタリなアプリケーションソフトウェアに対してもリターンアドレス書換え攻撃への対策を可能とするセキュアプロセッサ SAFFRON を実現する．具体的には，関数の出入口で実行される命令の対称性に着目し，リターンアドレスの保存と比較の処理を行うように命令の動作を拡張する．これにより，ユーザ空間で動作する全てのソフトウェアに対してリターンアドレス書換え攻撃を完全に防ぐことを可能とする．SAFFRON は，ハードウェアエミュレータである QEMU [10] を用いて IA-32 アーキテクチャの機能を拡張することで実装した．また，SAFFRON の有効性を検証するために，SAFFRON に対応すべく Linux カーネル（以降，Linux SAFFRON）を実装したのでそれについて述べる．さらに，テストプログラムを用いてリターンアドレス書換え攻撃が検知可能であるか検証したのでこれについても述べる．

以下，本論文では，2 章で本論文の関連研究について述べ，3 章で SAFFRON を用いたリターンアドレス書換え攻撃検知について述べ，4 章で実装について述べる．その後，5 章で評価について述べ，6 章で考察について述べる．最後に 7 章で本論文をまとめる．

2. 関連研究

バッファオーバーフロー攻撃への対策として主に，コンパイラを用いたスタック保護 [1]，OS を用いたメモリ空間のランダマイズ（ASLR）やデータ実行防止 [2], [3] がある．しかし，これらの対策は攻撃を困難にすることは可能であるが，ROP 攻撃 [6] によって対策を回避することが可能である．ROP 攻撃への対策としては，ホスト型侵入検知システムの異常検知 [5] が有効である．侵入検知システムの異常検知システムは，正常な動作時のシステムコールの実行順序と関数呼出し順序を解析したモデルを基にして，攻撃を受けた際のソフトウェアの異常な動作を検知する．しかし，ホスト型侵入検知システムの異常検知を回避する攻撃として mimicry attack [7] や mimicry attack を発展させた攻撃 [8]（以降，これらを総称して mimicry attack とする）が存在する．mimicry attack は，システムコールの実行順序や関数呼出し順序をソフトウェアの正常な動作のように模倣しながら攻撃を行うため，従来手法では異常を検知することができない．

以上のようなバッファオーバーフロー脆弱性を悪用する攻撃は，リターンアドレスを書き換えることでソフトウェアの制御フローを開発者の意図しない制御フローに変更する．そのため，リターンアドレスの書換えを検知すればリターンアドレスを書き換えるすべての攻撃を防ぐことが可能となる．リターンアドレスの書換えを検知する手法として，ROPdefender [11]，Branch Regulation [12]，スタック領域のリターンアドレス検査によるスタック偽装攻撃検知手法 [9] が存在する．これらの手法は，関数呼出し時にリターンアドレスをコールスタックとは別の領域に複製し，関数終了時に現在のリターンアドレスと複製したリターンアドレスを比較することで，リターンアドレスの書換えを検知する．ROPdefender はプロセス VM を用いた対策であり，Branch Regulation はハードウェアレベルでの対策である．そのため，攻撃検知のためにソフトウェアを再コンパイルする必要がない．しかし，直近のリターンアドレスの比較のみを行うため，mimicry attack により関数の実行順序が模倣された場合に攻撃を検知できない．

一方，スタック偽装攻撃検知手法は，比較時に全てのリターンアドレスを検査するため，mimicry attack を検知することが可能である．しかし，コンパイラを用いてコードを挿入する手法であるため，プロプライエタリなソフトウェアに対して適用することができない．そこで，SAFFRON では，ハードウェアレベルでリターンアドレスの書換えを検知することによって，ユーザ空間で動作する全てのソフトウェアに対するリターンアドレス書換え攻撃を完全に防ぐことを可能とする．

3. SAFFRON を用いたリターンアドレス書換え攻撃検知

3.1 概要

本章では，2 章で述べたスタック偽装攻撃検知手法を，ハードウェアレベルで実現するための設計について述べる．具体的には，IA-32 アーキテクチャをベースとし，当該プロセッサに以下の機能を追加することで実現する．

- リターンアドレスを保存・比較する処理のための命令拡張．

© 2015 Information Processing Society of Japan

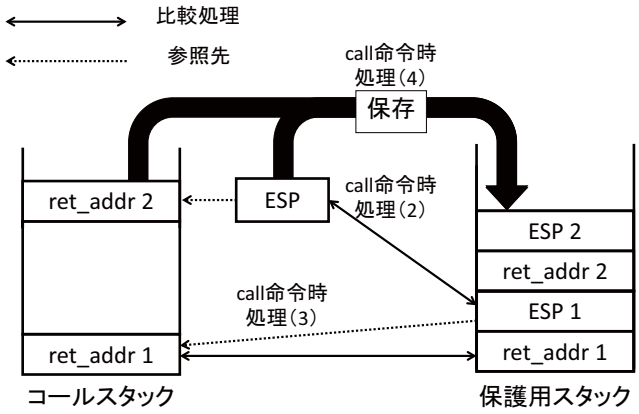

図 1 制御データの保存処理時の各スタックの様子

- プロセス・スレッドごとのコールスタックを管理するためのレジスタと命令の追加.

3.2 リターンアドレスの保存・比較処理

SAFFRON におけるリターンアドレス書換え攻撃検知は，以下の 2 つの要件を満たすことで可能となる.

- 関数呼出しを行う call 命令に，保護用スタックにリターンアドレスとリターンアドレスが格納されたメモリアドレスを制御データとして保存する処理を追加する.
- 関数を終了する ret 命令に，保護用スタックの制御データに基づいて探索，比較することで全てのリターンアドレスを検査する.

制御データを保存する保護用スタックは，攻撃者が書き換えることのできない領域に確保する必要がある．SAFFRON では，設計と実装の簡易化のためにプロセッサ内に保護用スタック用の領域があると仮定する．この領域は，QEMU のメモリ領域とすることで対応する.

3.2.1 call 命令の拡張

call 命令の処理を図 1 に示し，拡張した call 命令の処理を以下に示す．以下の手順は図中の番号と対応している．なお，手順（1），（5）は元々の call 命令の処理である．また，手順（2），（3）は 3.4 節で述べる重要な例外への対処となっている.

(1) EIP レジスタの値（リターンアドレスとなる値）をコールスタックに保存する.
(2) 現在の ESP レジスタ（リターンアドレスが格納されたアドレス）の値と保護用スタックの先頭にある制御データの ESP レジスタの値を比較する.
  - 保護用スタックが空の場合は，手順（3）に処理を移す.
  - 現在の ESP レジスタの値が保護用スタックの制御データの ESP レジスタの値よりも小さい場合（現在の ESP レジスタの値がスタックの上位を指している場合）は，手順（3）に処理を移す.

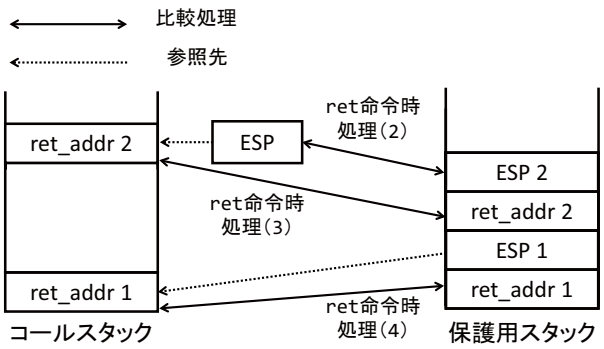

図 2 制御データの比較処理時の各スタックの様子

  - 現在の ESP レジスタの値が保護用スタックの制御データの ESP レジスタの値以上の場合（現在の ESP レジスタの値がスタックの下位を指している場合）は，保護用スタックの先頭にある制御データをポップする．その後，手順（2）の処理を行う.
(3) 保護用スタックの先頭にある制御データの ESP レジスタが指すコールスタックのリターンアドレスと，保護用スタックの先頭にある制御データのリターンアドレスを比較する.
  - 保護用スタックが空の場合は，手順（4）に処理を移す.
  - 値が一致した場合は，手順（4）に処理を移す.
  - 値が一致しない場合は，保護用スタックの先頭にある制御データをポップする．その後，手順（3）の処理を行う.
(4) 現在の制御データを保護用スタックに保存する.
(5) call 命令のオペランドで指定されたアドレスに処理を移す.

3.2.2 ret 命令の拡張

ret 命令の処理を図 2 に示し，拡張した ret 命令の処理を以下に示す．以下の手順は図中の番号と対応している．なお，手順（1），（6），（7）は元々の ret 命令の処理である．また，手順（2）は 3.4 節で述べる重要な例外への対処となっている.

(1) ESP レジスタが指すコールスタックのリターンアドレスを EIP レジスタにポップする.
(2) 現在の ESP レジスタの値と保護用スタックの先頭にある制御データの ESP レジスタの値を比較する.
  - 値が一致した場合は，手順（3）に処理を移す.
  - 現在の ESP レジスタの値が保護用スタックの制御データの ESP レジスタの値よりも大きい場合（現在の ESP レジスタの値がスタックの下位を指している場合）は，保護用スタックの先頭にある制御データをポップする．その後，手順（2）の処理を行う.
  - 現在の ESP レジスタの値が保護用スタックの制御データの ESP レジスタの値よりも小さい場合（現在の ESP レジスタの値がスタックの上位を指している

© 2015 Information Processing Society of Japan

表 1　追加した特権命令

ニーモニック	説明
LPID	EAX レジスタの値を PIDR に格納する.
LTID	EAX レジスタの値を TIDR に格納する.
CPS	EAX レジスタの値をプロセス識別子，EDX レジスタの値をスレッド識別子として保護用スタックを確保する.
DPS	EAX レジスタの値をプロセス識別子，EDX レジスタの値をスレッド識別子として保護用スタックを破棄する.
RPS	EAX レジスタの値を子プロセスのプロセス識別子，EDX レジスタの値を子プロセスのスレッド識別子，EBX レジスタの値を親プロセスのプロセス識別子，ECX レジスタの値を親プロセスのスレッド識別子として，親プロセスの保護用スタックを子プロセスの保護用スタックに複製する.

場合）は，攻撃を受けたと判断し，CPU の例外を発生させる.
- 保護用スタックが空の場合は，攻撃を受けたと判断し，CPU の例外を発生させる.

(3) 手順（1）のリターンアドレスと保護用スタックの先頭に保存した制御データのリターンアドレスを比較する.
- 値が一致した場合は，手順（4）に処理を移す.
- 値が一致しない場合は，攻撃を受けたと判断し，CPU の例外を発生させる.

(4) 前の関数呼出しの際に保護用スタックに保存した制御データの ESP レジスタの値が指すコールスタックのリターンアドレスと，その ESP レジスタと対となる保護用スタックに保存した制御データのリターンアドレスの値を比較する.
- 保護用スタックの底まで比較が終わった場合は，手順（5）に処理を移す.
- 値が一致した場合は，手順（4）の処理を行う.
- 値が一致しない場合は，攻撃を受けたと判断し，CPU 例外を発生させる.

(5) 保護用スタックの先頭にある制御データをポップする.

(6) ret 命令にオペランドがある場合は，オペランドに指定された数だけコールスタックのデータをポップする.

(7) 手順（1）で EIP レジスタにポップしたリターンアドレスの位置に処理を移す.

3.3　プロセス・スレッドごとの保護用スタック管理

コールスタックは，プロセス・スレッドごとに作成される．そのため，保護用スタックもプロセス・スレッドごとに管理する必要がある．OS は，プロセス・スレッドを一意の識別子を用いて識別している．これらの識別子は，OS の内部データであるため，ハードウェアからプロセス・スレッドを識別できない．そこで，OS との協調により，これを解決する．具体的には，プロセス・スレッド識別用のレジスタ PIDR・TIDR を追加し，OS がこれらのレジスタに現在実行しているプロセス・スレッドの識別子を格納することで解決する．また，これらのレジスタの操作は命令を新たに追加することで行う．これにより，ハードウェアから現在実行中のプロセス・スレッドを識別可能になり，保護用スタックを一意に定めることができる.

実装するにあたって追加した特権命令を表 1 に示す．LPID 命令・LTID 命令は，OS がコンテキストスイッチを行う際に実行することで，PIDR・TIDR に現在実行しているプロセス・スレッドの識別子を格納する．これにより，ハードウェアからプロセス・スレッドの識別が可能となり，プロセス・スレッドごとの保護用スタックの管理が可能となる.

CPS 命令は，プロセス・スレッドの生成時に保護用スタックを作成する命令である．この命令は，OS がプロセス・スレッドを生成する際に，プロセス・スレッドの識別子をオペランドとして指定することで，対応する保護用スタックを作成する.

DPS 命令は，プロセス・スレッドの終了時に保護用スタックを破棄する命令である．この命令は，OS がプロセス・スレッドを終了する際に，プロセス・スレッドの識別子をオペランドとして指定することで，対応する保護用スタックを破棄する.

RPS 命令は，UNIX 系 OS の場合に見られる，親プロセスの複製を生成した後に子プロセスのコールスタックを作成するという実装に対応するための命令である．この命令は，親プロセスの保護用スタックの複製を行う．OS は，プロセス・スレッドを複製する際に，親プロセス・子プロセスの識別子をオペランドとして指定することで，スタックの複製を行う.

3.4　重要な例外

3.4.1　課題

リターンアドレスの保存処理は，関数を呼び出す命令である call 命令を拡張することで実現できる．また，リターンアドレスの比較処理は，関数終了時の命令である ret 命令を拡張することで実現できる．しかし，Linux などの実際の OS やアプリケーションを動作させる上で次の 2 つの処理に対して特別な対応が必要となることが明らかとなった.
- zero length call による現在実行中のメモリアドレスの取得.
- setjmp/longjmp による大域ジャンプ.

これらの技術が利用されると call 命令と ret 命令の対称性が損なわれ，比較処理の際に偽陽性が発生する.

© 2015 Information Processing Society of Japan

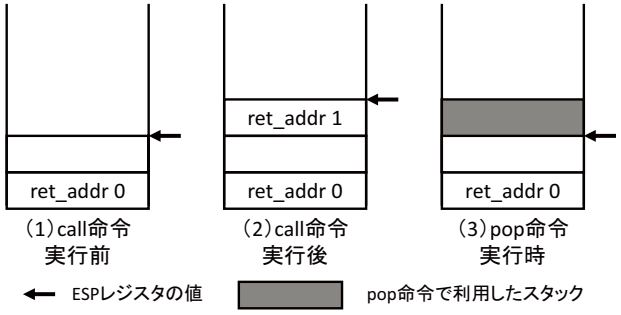

図 3 zero length call を用いた際のコールスタックの様子

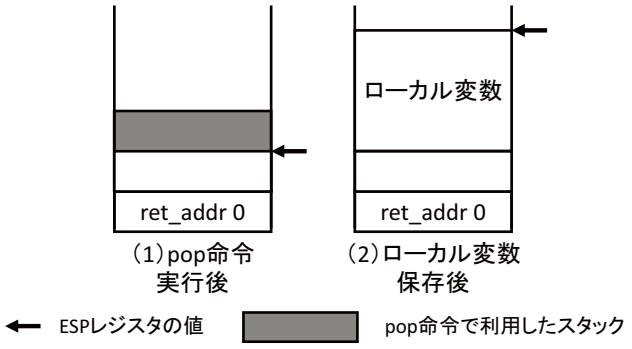

図 4 zero length call を用いた後のコールスタックの様子

zero length call

zero length call は IA-32 アーキテクチャにおける現在の命令アドレスを取得するための技術である．この技術は，call 命令のジャンプ先を次の命令アドレスとし，pop 命令を実行する．これにより，call 命令でコールスタックに保存した，リターンアドレスを取得する．zero length call を利用した後に以下の 2 つの事象が発生する．

(1) 保護用スタックの制御データに対応するリターンアドレスが破棄される．
(2) ローカル変数によってリターンアドレスが上書きされる．

事象 (1) の例として，zero length call を用いた際のコールスタックの様子を図 3 に示す．図 3 の処理を以下に示す．なお，以下の手順は図中の番号と対応している．

(1) call 命令を実行する直前のコールスタックの状態である．
(2) call 命令により，ret_addr 1 のリターンアドレスがコールスタックに保存された状態である．
(3) pop 命令を実行し，コールスタックに保存された ret_addr 1 を取り出した状態である．

このように，図 3 の ret_addr 1 は，ret 命令で利用されない．しかし，保護用スタックには ret_addr 1 に対応する制御データが保存された状態となり，コールスタックと保護用スタックに差異が生じるため，偽陽性が発生する．

事象 (2) の例として，zero length call を用いた後のコールスタックの様子を図 4 に示す．図 4 の処理を以下に示す．なお，以下の手順は図中の番号と対応している．

(1) zero length call を用いてリターンアドレスを取り出した状態である．
(2) ローカル変数がスタックに積まれ，リターンアドレスを格納していたメモリが上書きされた状態である．

このように，zero length call を実行した後にローカル変数がコールスタックにプッシュされると，リターンアドレスが上書きされた状態となるため，偽陽性が発生する．

setjmp/longjmp

C 言語のライブラリ関数である setjmp/longjmp 関数は，setjmp 関数で現在のコールスタックの状態を保存し，longjmp 関数で保存したコールスタックの状態へ戻す処理を行う．これらの関数は，ソフトウェアの例外処理などに用いられる．これらの関数が実行された場合においても，zero length call と同様の事象が発生する．

3.4.2 解決策

3.4.1 項で述べた処理は，スタックが戻される際に ESP レジスタの値が増加する．また，ハードウェアは call 命令と ret 命令の実行時に ESP レジスタの値の増加を検知することが可能である．そこで，call 命令と ret 命令の実行時に ESP レジスタの値を比較し，比較結果に基づいて制御データを破棄することで，これらの処理に対応する．

3.4.1 項の事象（1）は，現在の ESP レジスタの値が指すコールスタックの先頭よりもスタックの上位にあるデータは既に使用されないデータ（図 3 中の pop 命令で利用されたスタック）であると判断し，対応する保護用スタックの制御データを破棄することで対応が可能である．この処理を call 命令の保存処理と ret 命令の比較処理で行う．また，call 命令の保存処理では，制御データの先頭にある ESP レジスタの値と現在の ESP レジスタの値が一致した場合についてもコールスタックが戻されたものであると判断し，保護用スタックの制御データを破棄する．これらの処理は，3.2.1 項の call 命令の拡張における手順（2）と，3.2.2 項の ret 命令の拡張における手順（2）に該当する．

3.4.1 項の事象 (2) は，ローカル変数をコールスタックにプッシュすることでリターンアドレスが上書きされた後に call 命令が実行される際に問題となる．これは，事象（1）への対応では検知できない．そこで，call 命令の際にコールスタックのリターンアドレスが書き換えられていないか検査し，書き換えられていた場合に保護用スタックの制御データを破棄することで対応する．この検査処理をリターンアドレスが書き換わっていない箇所が見つかるまで繰り返す．保護用スタックの制御データが全て破棄された場合は，最初の関数呼出しまでスタックが戻されたと判断し，制御データの保存処理を行う．この処理は，3.2.1 項の call 命令の拡張における手順（3）に該当する．これらの処理を行うことで zero length call や setjmp/longjmp 関数のように ret 命令が実行されない場合に対応することが可能で

© 2015 Information Processing Society of Japan

ある．

4. 実装

3 章の機構を実現するために SAFFRON と Linux SAFFRON を実装した．SAFFRON は，QEMU 1.0 をベースとして実装した．本章では，これらの実装について述べる．

4.1 SAFFRON の実装

QEMU は，ゲスト OS のコードを動的にホスト OS で実行可能な命令に変換することでエミュレーションを行う．変換は，中間コードを経由することで，様々なアーキテクチャでのエミュレーションを容易にしている．SAFFRON は，ゲストコードを中間コードに変換する target-i386/translate.c ファイル内の disas_insn 関数内に処理を追加することで命令拡張や命令追加を行う．また，target-i386/cpu.h ファイル内の CPUX86State 構造体のメンバに PIDR・TIDR を追加することでレジスタを追加した．保護用スタックを管理するデータ構造は，以下の情報を持つ．

- プロセス識別子を格納するメンバ
- スレッド識別子を格納するメンバ
- 保護用スタックの先頭を指すポインタ
- 保護用スタックの位置を指すポインタ

SAFFRON は，これらの情報をメンバとして持つ構造体を QEMU 内に malloc 関数で確保し，リスト構造で管理する．

4.2 Linux SAFFRON の実装

Linux SAFFRON は以下の機能にコードを追加することで SAFFRON に対応させた．

4.2.1 プロセス・スレッドの生成

Linux のプロセス・スレッドの生成は，まず，最初のプロセスである init プロセスを作成する．その後 Linux は，fork システムコールで親プロセスの複製を生成し，生成したプロセスで exec システムコールを用いて新たに別のソフトウェアを実行することで子プロセスを実行する．そのため，保護用スタックは次のように作成する．Linux SAFFRON は，fork システムコールを用いて子プロセスを生成する際に RPS 命令を用いて親プロセスの保護用スタックを複製する．その後，exec システムコールを用いて新たに実行するソフトウェアにメモリを書き換える際に複製した保護用スタックを DPS 命令で破棄し，CPS 命令を用いて新たな保護用スタックを作成することで，新たに実行するソフトウェアの保護用スタックを作成する．

具体的には，RPS 命令を kernel/fork.c ファイルに含まれる copy_process 関数内で実行する．この関数は，プロセス・スレッドを複製する．また，RPS 命令は，作成するプロセス・スレッドの識別子（task_struct 構造体のメンバ tgid・pid）の値をそれぞれ EAX レジスタ，EDX レジスタに格納し，複製元のプロセスの識別子の値を EBX レジスタ，ECX レジスタに格納した後に実行する．その後，DPS 命令と CPS 命令を fs/binfmt_elf.c ファイルに含まれる load_elf_binary 関数内で実行する．この関数内で新たに作成したプロセスにプログラムをロードする処理が行われる．また，DPS 命令と CPS 命令は，新たにプログラムをロードするプロセス・スレッドの識別子の値をそれぞれ EAX レジスタ，EDX レジスタに格納した後に実行する．この処理のために追加したコードは，22 行である．

4.2.2 プロセス・スレッドの終了

Linux のプロセス・スレッドの終了は，exit システムコールで実行される．そのため，Linux SAFFRON はメモリを開放する処理に保護用スタックを破棄する DPS 命令を追加する．

具体的には，DPS 命令を kernel/exit.c ファイルに含まれる release_task 関数内で実行する．この関数は，プロセス・スレッドの終了処理の最終段階でメモリを開放する処理を行う．また，DPS 命令は，終了するプロセス・スレッドの識別子の値をそれぞれ EAX レジスタ，EDX レジスタに格納した後に実行する．この処理のために追加したコードは，22 行である．

4.2.3 コンテキストスイッチ

Linux のプロセス・スレッドの切り替えは，全てコンテキストスイッチ関数を用いて行われる．そのため，Linux SAFFRON はコンテキストスイッチ関数に LPID・LTID 命令を追加する．

具体的には，LPID・LTID 命令を kernel/sched/core.c ファイルに含まれる context_switch 関数内から呼ばれる switch_to マクロの直前で実行する．switch_to マクロの直前で実行する理由は，このマクロを実行することで実行するプロセス・スレッドのメモリ領域を切り替えるためである．また，LPID・LTID 命令は，次に実行するプロセス・スレッドの識別子を格納している変数のメモリアドレスをオペランドとして指定する．この処理のために追加したコードは，17 行である．

5. 評価

SAFFRON の有効性を検証するために，SAFFRON に対応するように手を加えた Linux SAFFRON を用いて動作検証を行った．本章では，Linux SAFFRON の実装と自作のテストプログラムを用いた動作検証の結果について述べる．

5.1 環境

評価環境を表 2 に示す．Buildroot は，Makefile とパッチから構成され，一括して組込み Linux 環境を構築するシステムである．BUildroot が構築する Linux 環境は，カーネルイメージとルートファイルシステムからなる．ルートファイルシステムは，主な標準 UNIX コマンドの機能

© 2015 Information Processing Society of Japan

表 2　評価環境

項目	内容
OS（カーネル）	Linux SAFFRON（Linux 3.10.5）
標準 C ライブラリ	uClibc 0.9.33
BusyBox	Version 1.22.1
Buildroot	Version 2014.08

を提供するアプリケーションである BusyBox が含まれる．Linux 環境で一般的に使用される標準 C ライブラリ glibc は，call 命令と ret 命令の対称性がないアセンブリコードを含んでいる．そのため，call 命令と ret 命令に対称性がある uClibc を用いて評価環境を構築した．

5.2 節で述べるテストプログラムは，BusyBox に含まれるシェルプログラム上で動作させた．

5.2　内容

自作のテストプログラムを用いて偽陽性・偽陰性が発生しないことを確認する．検証項目を以下に示す．

(1) 現在のスタックフレームのリターンアドレスを書き換えるテスト
このテストにより，バッファオーバーフロー攻撃などによるリターンアドレス書換え攻撃の検知が可能であることを確認する．

(2) 1 つ前のスタックフレームのリターンアドレスを書き換えるテスト
このテストにより，関数の呼出し順序の模倣する mimicry attack の検知が可能であることを確認する．

(3) スレッドを用いたテスト
clone システムコールを用いてスレッドを生成し，プロセス・スレッドの識別が可能であるかを確認する．さらに，生成したスレッドに対して（1）・（2）のテストを行った．このテストにより，スレッドに対してもリターンアドレス書換え攻撃検知が有効であることを確認する．

(4) setjmp/longjmp 関数を用いたテスト
setjmp/longjmp 関数を用いた際に，3.4 節で述べた処理で偽陽性が発生しないことを確認する．さらに，longjmp 関数実行後に（1）・（2）のテストを行った．このテストにより，setjmp/longjmp 関数を用いた場合に対してもリターンアドレス書換え攻撃検知が有効であることを確認する．

5.3　結果

5.2 節のテストを行った結果を以下に示す．（1）を実行した結果，リターンアドレスを書き換えた後，最初の ret 命令でリターンアドレス書換えが検知可能であることを確認した．（2）を実行した結果，リターンアドレスを書き換えた後，最初の ret 命令でリターンアドレス書換えが検知可能であることを確認した．

（3）を実行した結果，子スレッド・親スレッドともに偽陽性が発生することはなかった．次に，子スレッドに対して（1），（2）と同様のテストを行った結果，上述の（1），（2）に対する結果と同様になり検知可能であることを確認した．

（4）の setjmp/longjmp 関数を使用するテストプログラムを実行した結果，偽陽性が発生することはなかった．次に，setjmp/longjmp 関数を実行した後に（1），（2）と同様のテストを行った結果，上述の（1），（2）に対する結果と同様になり，検知可能であることを確認した．

6.　考察

call 命令と ret 命令の対称性が損なわれるケースへの対応について考察する．ROPdefender[11] によって以下の 3 種類に分類されている．

ケース 1：ret 命令が実行されない場合

call 命令で関数が呼び出されるが，ret 命令でリターンアドレスを使用しない場合がある．例えば，C 言語のライブラリ関数である setjmp/longjmp 関数や zero length call などが利用された際に発生する．

ケース 2：call 命令を実行せずに関数を呼び出す場合

push 命令を用いてリターンアドレスとなる値をコールスタックに保存し，ret 命令で関数を終了する場合がある．例えば，Unix のシグナルや動的リンカの遅延バインドが利用された際に発生する．

ケース 3：リターンアドレスが書換えられる場合

ソフトウェア実行時にコールスタックのリターンアドレスを書き換える処理がある．例えば，C++の例外処理がこれにあたる．

ケース 1 への対応は，3.4 節で述べた解決策によって実現できる．しかし，拡張した call 命令の手順（3）によって保護用スタックの制御データを破棄するため，攻撃者がリターンアドレスを書き換えた後に call 命令を実行した場合に攻撃の検知が遅れる可能性がある．例としてプログラムを図 5 に示す．このプログラムでは，funcB 関数内の 15 行目でリターンアドレスを書き換え，その後 16 行目で funcD 関数を call 命令で呼び出す．このため，拡張した call 命令の手順（3）により funcB 関数に対応する制御データは破棄される．したがって，4 行目の printf 関数や funcD 関数の ret 命令では，funcB 関数に対応する制御データの比較が行われない．その後，funcB 関数が終了し，リターンアドレスが書換えられたことによって funcC 関数へ処理が移る際に，拡張した ret 命令の手順（2）で攻撃であると検知する．この検知のタイミングは，ROPdefender や BR と同様のタイミングとなる．また，拡張した call 命令の手順（3）により偽陰性が発生する可能性があるため，今後検討する必要がある．

ケース 2 とケース 3 への対応は，保護用スタックに制御

© 2015 Information Processing Society of Japan

```
 1 #include <stdio.h>
 2
 3 funcD() {
 4     printf("funcD was called\n");
 5 }
 6
 7 funcC() {
 8     printf("unreachable code\n");
 9     while(1);
10 }
11
12 funcB() {
13     int a[1];
14     void *p = funcC;
15     a[5] = (int)p; // a[5] is funcB's return address
16     funcD();
17 }
18
19 funcA() {
20     funcB();
21     printf("funcB was returned\n");
22 }
23
24 int main() {
25     funcA();
26 }
```

図 5 リターンアドレス攻撃検知が遅くなるケース

データを格納する特権命令を新たに追加することで対応が可能であると考えられる．ケース 2 は，リターンアドレスとしてコールスタックにデータを保存する際に，新たに追加する命令を実行することで対応が可能であると考えられる．また，ケース 3 へは，ケース 2 への対応の命令と，保護用スタックから制御データを破棄する特権命令を新たに追加することで対応が可能であると考えられる．ケース 3 はリターンアドレスを書き換える際に，まず，コールスタックから破棄するリターンアドレスの数だけ保護用スタックから制御データを破棄する．その後，書き換える対象のリターンアドレスを破棄した後に，ケース 2 の対応と同様に保護用スタックに制御データを格納する命令を実行することで対応が可能であると考えられる．

7. おわりに

本論文では，ハードウェアエミュレータである QEMU を用いた IA-32 アーキテクチャにおけるリターンアドレス書換え攻撃検知機構 SAFFRON について述べた．また，SAFFRON を利用するために Linux に手を加えた Linux SAFFRON についても述べた．リターンアドレスを利用する call 命令と ret 命令を拡張し，プロセス・スレッドの管理を行う命令を追加することで，リターンアドレス書換え攻撃の検知が可能であることを確認した．また，call 命令と ret 命令の対称性が損なわれるケースについての対応策についても述べた．

今後の課題として，call 命令と ret 命令の対称性が損なわれる場合への対応策の実装がある．例外ケースへの対応を行い，偽陰性・偽陽性が発生しないことを十分にテストする必要がある．

参考文献

[1] Microsoft: /GS (Buffer Security Check), http://msdn.microsoft.com/en-us/library/8dbf701c.aspx (2015).
[2] Team, T. P.: Homepage of The PaX Team, http://pax.grsecurity.net/ (2015).
[3] Molnar, U.: Index of /mingo/exec-shield, http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf (2004).
[4] Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156–168 (2001).
[5] 阿部洋丈，大山恵弘，岡　瑞起，加藤和彦：静的解析に基づく侵入検知システムの最適化，情報処理学会論文誌,, Vol. 45, No. 3, pp. 11–20 (2004).
[6] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86), *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 552–561 (2007).
[7] Wagner, D. and Soto, P.: Mimicry attacks on host-based intrusion detection systems, *Proceedings of the 9th ACM conference on Computer and Communications Security*, pp. 255–264 (2002).
[8] Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Automating mimicry attacks using static binary analysis, *Proceedings of the 14th conference on USENIX Security Symposium*, Vol. 14, pp. 161–176 (2005).
[9] 冨永悠生，樫山武浩，瀧本栄二，桑原寛明，毛利公一，齋藤彰一，上原哲太郎，國枝義敏：コールスタックの制御データ検査によるスタック偽装攻撃検知，情報処理学会論文誌，Vol. 53, No. 9, pp. 2075–2085 (2012).
[10] Bellard, F.: QEMU, a fast and portable dynamic translator, *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 41–46 (2005).
[11] Davi, L., Sadeghi, A.-R. and Winandy, M.: ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 40–51 (2011).
[12] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N. and Ponomarev, D.: Branch Regulation: Low-overhead Protection from Code Reuse Attacks, *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 94–105 (2012).

© 2015 Information Processing Society of Japan