Vol.2015-OS-132 No.13
2015/2/27

リリースされたバイナリに適用するスタックベース BoF 攻撃緩和技術の試作と評価 (ver.2)

A Prototyping and Evaluation of Prevention/Mitigation of Released Binaries against Stack-based Buffer Overflow Attacks (ver.2)

齋藤 孝道* 上原 崇史† 金子 洋平†
角田 佳史† 堀 洋輔† 馬場 隆彰* 宮崎 博行*
Takamichi SAITO* Takafumi UEHARA† Yohei KANEKO†
Yoshifumi SUMIDA† Yosuke HORI† Takaaki BABA* Hiroyuki MIYAZAKI*

あらまし 脆弱性の種類を識別するための共通の脆弱性タイプの一覧である CWE において，メモリ破損脆弱性の中に分類される，いわゆる Stack-based Buffer Overflow は，現在でも NVD での報告が絶えない脆弱性の一つである．現在までに Stack-based Buffer Overflow に対する様々な攻撃対策技術が開発されているが，その多くがコンパイル時における適用を求められる．よって，開発者から既にリリースされているソフトウエアに対して，その種の攻撃対策技術を追加で適用する場合には，ソースコードを用意し再コンパイルしなければならない．そこで，本論文では，Stack-based Buffer Overflow 攻撃を緩和する新たな手法を，ソースコードがないバイナリに対しても適用可能とするように ELF Loader において実現し，その評価を行う．

キーワード Stack-based Buffer Overflow，ELF Loader，脆弱性緩和・対策

1 はじめに

脆弱性の種類を識別するための CWE（Common Weakness Enumeration）[1]において，メモリ破損脆弱性の中に分類される，いわゆる Buffer Overflow は，現在でも NVD (National Vulnerability Database) [2]で，報告が絶えない脆弱性の一つである．

2014 年のマイクロソフトの調査[3]によると Stack-based Buffer Overflow[4]は，2007 年には 54.2%を占めていたが，2013 年には 5%まで減少したと報告されている．また，Stack-based Buffer Overflow の減少の理由は，コンパイラにおける/GS や SafeSEH 等の対策技術が機能していることや開発時の静的な分析ツールの精度が向上による貢献であると結論付けられている．しかし，Stack-based Buffer Overflow の脅威がなくなったわけではなく，NVD においても，Buffer Overflow の中で，最も数多く報告されている．

現在，Stack-based Buffer Overflow に対する様々な対策技術があるが，その多くがコンパイル時における適用が求められる．

* 明治大学, 214-8571 神奈川県川崎市多摩区東三田 1-1-1,
Meiji University 1-1-1, Higashimita, Tama-ku Kawasaki-shi, Kanagawa, 214-8571, Japan
† 明治大学大学院, 214-8571 神奈川県川崎市多摩区東三田 1-1-1,
Graduate School of Meiji University 1-1-1, Higashimita, Tama-ku Kawasaki-shi, Kanagawa, 214-8571, Japan

我々の調査によると，3 つの Linux ディストリビューションにおける Stack-based Buffer Overflow に対する対策は，高々50%ほどしか適用されていないことが分かった．

そのような背景がある一方，コンパイラによる対策での場合，開発後リリースされたソフトウエアに対して防御・攻撃緩和機能を追加で適用する場合には，保護対象のソフトウエアのソースコードを用意し，再コンパイルしなければならない．その上で，様々な検査を行う必要があり，対策技術によっては，生成された実行コードのサイズが増える場合もある．

そこで，本論文では，対策技術が適用されていない，もしくは，適用できないソフトウエアに対し，保護対象とするソフトウエアの実行ファイルに修正を加えることのなく，Stack-based Buffer Overflow への対策方式を提案する．実現方法として，実行コードをロードし実行するローダーにおいて，新たな対策技術を導入する．本論文では，導入した対策技術の評価も行う．

提案方式のローダー（以下，ELF Loader と呼ぶ）としては，C 言語で書かれたプログラムかつ GCC（GNU Compiler Collection）でコンパイルされた Linux ELF（Executable and Linkable Format）形式[5]のバイナリを 32bit の Linux OS にて利用する環境で動作する．

2 関連技術

2.1 本論文で対象とする攻撃について

Buffer Overflow （CWE-120）[6]とは，メモリ破損

（CWE-119）の一種で，入力データを検査しないプログラムの脆弱性によって，プログラム内でバッファとして確保している範囲を超えて，メモリ領域にデータが書き込まれてしまう現象もしくは脆弱性のことである．この脆弱性を悪用した攻撃を Buffer Overflow 攻撃という．攻撃者はバッファサイズ以上のデータでバッファを溢れさせ，プログラムの制御フローを攻撃者の意図した動作に変えるようにメモリの内容を書き換える．

2.1.1 Stack-based Buffer Overflow 攻撃

Stack-based Overflow（CWE-121）[4]とは，スタック領域に確保されたバッファで Buffer Overflow を引き起こす脆弱性である．スタック領域では，関数ポインタやフレームポインタ（saved %ebp），リターンアドレス（saved %eip）が書き換え対象となる．この脆弱性を利用した攻撃を Stack-based Buffer Overflow 攻撃という．

図 1 は，main 関数から func 関数を呼び出した正常時の関数のスタックフレームを示す．

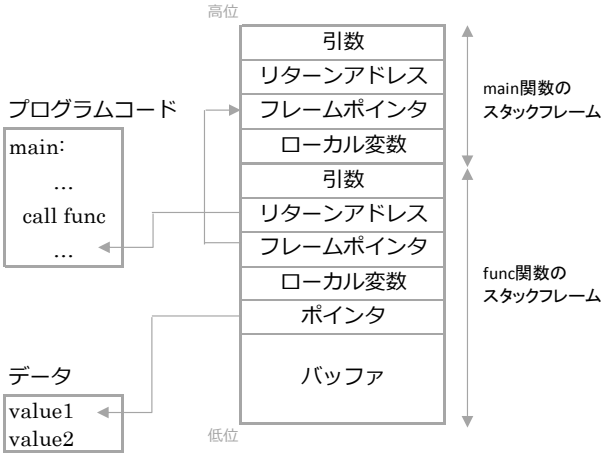


図 1　正常時のスタックメモリレイアウト

図 2 は，リターンアドレスを書き換え対象とした攻撃時のスタックメモリ領域を示す．リターンアドレスを書き換え対象とする攻撃として，Return-to-libc 攻撃[7]，Return-to-Register 攻撃[8]，Return-Oriented-Programming 攻撃[9]，Return-to-plt／Return-to-strcpy 攻撃[10]などがある．また，フレームポインタを書き換え対象とする攻撃として，フレームポインタ書き換え攻撃[11]，Off-by-one 攻撃[12]がある．いずれも，Stack-based Buffer Overflow 攻撃を，それぞれの攻撃のきっかけとしている点に注意されたい．

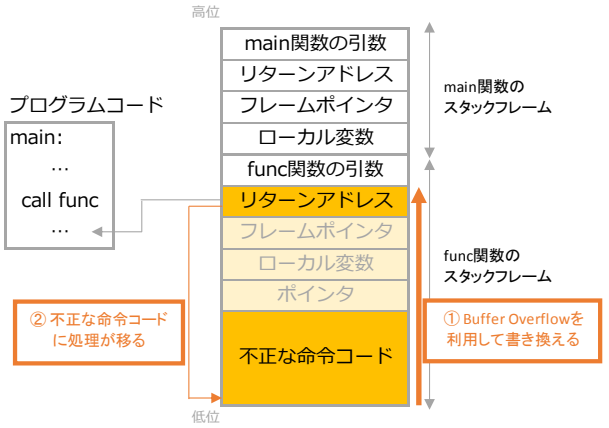


図 2　リターンアドレスを書き換え対象とした攻撃時のスタックメモリイメージ

2.1.2　既存の対策技術を回避する攻撃

2.1.2.1 Canary 偽装攻撃

Canary 偽装攻撃[13]とは，後述する SSP（Stack Smashing Protector．以降，SSP と呼ぶ）[14]適用時にフレームポインタの低位に挿入される canary 値を偽装することによって，Stack-based Buffer Overflow 攻撃を実現する攻撃のことである．実際に，Linux（Ubuntu14.04 Kernel 3.13.0-34-generic, gcc-4.9.1）は，プロセスが生成された時点での canary を使いまわしているということを我々は確認している．よって，例えば，親プロセスで生成された canary を何らかの方法で取得できれば，子プロセスでは，canary を偽造できる可能性がある．

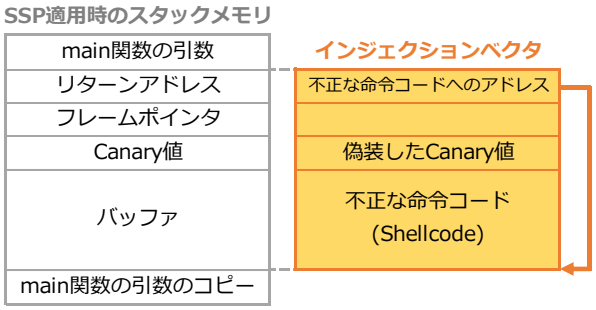


図 3　Canary 偽装攻撃時のスタックメモリイメージ

2.1.2.2 Return-to-Register 攻撃

Return-to-Register 攻撃[8]とは，ret 命令実行後にレジスタが指しているアドレスに不正な命令コードを挿入し，その上で，“そのレジスタ値に実行を移す命令群が格納されているアドレス”でリターンアドレスを書き換える攻撃のことである．

図 4 は，esp レジスタを利用した場合の Stack-based Buffer Overflow 攻撃手法を示す．main 関数の ret 命令実行後には，esp レジスタは引数のアドレスを指す．そこで，攻撃者は，引数とリターンアドレスのメモリ領域を，“不正な命令コード”及び“jmp %esp に対応するバイト列”のいずれもが格納されているアドレスで書き換える．main 関数の ret 命令実行時に，書き換えられたリターンアドレスである“jmp %esp に対応するバイト列”が格納されているアドレスを eip レジスタに pop する．す

ると，jmp %esp により，esp レジスタが指している不正な命令コードに実行が移る．また，jmp %esp 以外にも，call %esp, push %esp, ret, call eax などの命令も悪用できる．

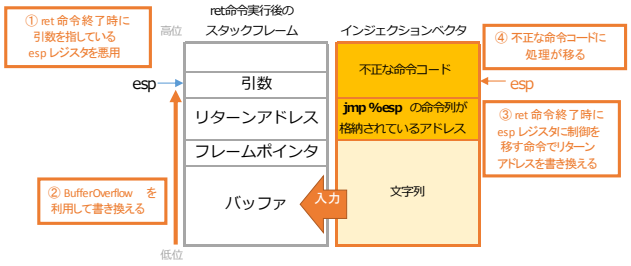


図 4　Return-to-Register 攻撃時のスタックメモリイメージ

2.1.2.3. Return-Oriented-Programming 攻撃

ROP (Return-Oriented Programming) 攻撃 [9]とは，ASLR によって配置アドレスがランダム化されないライブラリ関数や実行プログラムの命令コード（以降，ガジェットという）を組み合わせて shellcode などの不正な命令コードの代替とする攻撃のことである．ROP 攻撃の場合，ret 命令で終わる命令コード（以降，ROP ガジェットという）を組み合わせる（図 5 参照）．

ROP 攻撃の他に，jmp 命令で終わる命令コードを使用する JOP （Jump-Oriented Programming）攻撃[15]や ROP ガジェットと JOP ガジェットを使い分け，書式文字列の問題を利用する SOP（String-Oriented Programming）攻撃[16]もある．ROP 攻撃や JOP 攻撃は，データ領域におけるコード実行防止機能と ASLR を回避することが可能となる．更に，SOP 攻撃では，コード実行防止機能と ASLR，SSP（Stack Smashing Protector）[14][17][18]を回避することが可能となる．

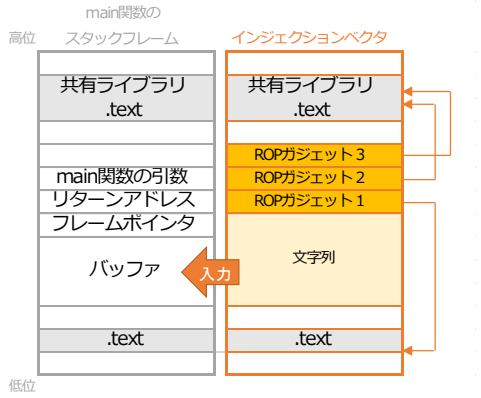


図 5　Return-Oriented-Programming 攻撃時のスタックメモリイメージ

2.1.2.4. Return-to-plt／Return-to-strcpy 攻撃

Return-to-plt 攻撃[10]とは，リターンアドレス書き換え攻撃の一種で，ライブラリ関数を呼び出す際に参照する間接ジャンプテーブルである“PLT（Procedure Linkage Table）領域（.plt セクション）へのアドレス”でリターンアドレスを書き換える攻撃のことである．また，その関数呼出しに必要な引数をスタックフレームに用意することで，攻撃者の意図したライブラリ関数を呼び出すことができる．

Return-to-plt 攻撃の一種である Return-to-strcpy 攻撃[10]とは，リターンアドレスを strcpy@plt へのアドレスで書き換え，strcpy 関数に必要なスタックフレームを用意することで，strcpy 関数を呼び出す攻撃のことである．NULL 文字（¥0x00）を含むアドレスを任意のメモリ領域に書き込む場合に利用する手法である（図 6 参照）．

このように.plt セクションを利用する攻撃手法は，ASCII-armor[19]を回避することが可能となる．

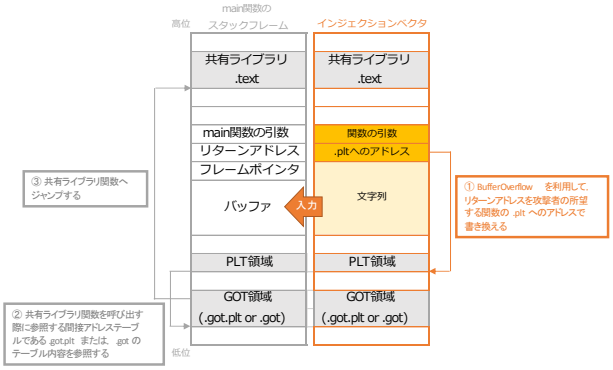


図 6　Return-to-plt／Return-to-strcpy 攻撃時のスタックメモリイメージ

2.2　類似対策技術

2.2.1　Stack Smashing Protector

GCC で導入されている SSP は，スタック領域上に配置されるリターンアドレス，フレームポインタ，引数，ローカル変数，及び，ポインタを保護する機能である．これは，IBM の StackGuard と呼ばれる保護機能を元に開発が行われた．GCC バージョン 4.1 以前では，パッチにて公開されていたが，バージョン 4.1 以降では標準機能とされている．

SSP は，スタック上にてフレームポインタより低位に canary と呼ばれる値を挿入し，その値が関数終了時に改竄されているかどうかを確認する．SSP を適用したバイナリの実行時，canary の改竄を検出した場合には，実行を停止する．

主に，SSP には，引数や関数ポインタの改竄を防ぐため，引数保護機能と関数ポインタ保護機能という機能がある[14]．これらを実現するため，実行コードの作成の際，ターゲット関数内で宣言されたバッファより低位に引数や関数ポインタを配置する．

前述の機能を適用するためには，SSP専用コードの命令列をコンパイル時に追加する必要がある．よって，後からSSPを適用する場合はソースコードが必要となる．また，Buffer Overflowの発生自体を防ぐことはできないという問題もある．更に，当然，追加の命令列が増えるので，実行ファイルのサイズ大きくなることも懸念される．

2.2.2　StackShield

StackShield[20]は，コンパイル時にリターンアドレスをデータ領域にコピーする 2000 年に提案されたコンパイラの対策技術である．GCC のパッチという形で提供されている[21]．関数終了直前にリターンアドレスのコピーを強制的にスタックに上書きする．関数終了時に上

書きされたリターンアドレスに実行が移る.

2.3 関連対策技術

この節では，提案方式に関連する対策技術について概観する.

2.3.1 コード実行防止機能

コード実行保護機能[17]とは，プログラム実行時に使用するスタック，ヒープ，.bss セクション及び.data セクション（テキストセクション），共有ライブラリの各領域上でのコードの実行を不可能にするハードウエア（NX bit/XD bit），及び，コンパイラ（execstack）による機能である[17][18].

2.3.2 ASLR

ASLR（Address Space Layout Randomization）[17]とは，アドレス空間配置のランダマイズ化のことで，プロセスのメモリ領域のマッピングを無作為に配置する OS による機能である. 特に, PIE（Position-Independent Executable）[17]形式でコンパイルしたバイナリを ASLR が有効な環境で実行する場合を Full-ASLR と呼ぶ. ここで，PIE とは，ELF 形式のバイナリにおけるテキストセグメントのアドレス空間配置のランダマイズ化を実現するファイル形式で，コンパイル時に GCC コンパイラのオプションで指定する.

2.3.3 RELRO

RELRO（RELocation Read-Only）[17]とは，ELF 形式のバイナリにおける各セクション，主にライブラリ関数のアドレスを保持する GOT（Global Offset Table）領域を読み取り専用にする動的リンカのセキュリティ機能である. GCC のオプションの一つとして提供されている.

遅延バインド有効の際の RELRO である Partial RELRO の場合には, 読み書き可能である.got.plt セクションが生成される. したがって，Partial RELRO の場合には，GOT 書き換え攻撃が行われる可能性がある.

遅延バインド無効の際の Full RELRO の場合には，.got.plt セクションは生成されず，データセグメント以外読み取り専用となる.

3 既存対策技術についての考察

既存の対策技術の多くは，ソフトウエアのコンパイル時に適用する対策となっている. したがって，コンパイル時に適用しなかった場合，後から，対策を適用しようとすると，あらかじめソースコードを入手していなければならない，かつ，追加で修正を加えるために，再コンパイルしなければならない. 折角，効果的な新しい対策技術が登場しても，それを適用するには，容易ではないケースも想定できる.

また，我々の調査によると，Stack-based Buffer Overflow に対する既存の対策技術である SSP が，適用されていないバイナリもあり，現状でも，その種の攻撃の余地を残しているといえる（表 1 参照）.

表 1 ディストリビューション別の SSP の対応状況

Dist. (32bit OS)	Num of Binaries	Canary found	No Canary found
CentOS 6.5	3626	44.76% (1623)	55.24% (2003)
Debian 7.5	631	66.51% (420)	33.44% (211)
Ubuntu 14.04	748	76.47% (572)	23.53% (176)

以上を鑑みて，既に，開発元からリリースされたソフトウエアに対して，各種の攻撃を防ぐ対策技術を適用するアプローチの必要性があるといえる. このアプローチでは，開発者が何らかの理由で対策を施すのを忘れ，更に，脆弱性をソフトウエアにいれ込んでしまっても，後に，運用のフェーズでも対策を取ることが可能となる.

4 提案方式について

4.1 概要

Stack-based Buffer Overflow 攻撃は，前述の通り，プログラム実行時，スタックにおけるリターンアドレスを書き換えることにより攻撃を完遂する. プログラム実行時に，提案方式により，リターンアドレスをスタックから下位のアドレス領域へ退避し，攻撃者によるリターンアドレス書き換えを不可能にした. さらに，実行コードに対して，この仕組みの導入をコンパイル時に行うのではなく，実行コードのロード時に，アプリケーションプログラマが定義した関数の先頭と終端にモジュール（以降，ret-shelter モジュールと呼ぶ）を挿入する. また，ret-shelter モジュールは，リターンアドレスが攻撃者により不正に書き換えられている場合も，それを検出する. ret-shelter モジュールの挿入は，User Land で稼働する ELF 形式の実行ファイルをロード・実行する Loader により行われる. その起動は別として，OS の ELF Loader とは独立して，実行コードを起動する. 以降，本論文で提案する ELF Loader を Ret-Shelter Loader と呼ぶ.

4.2 提案方式の詳細

以下に提案方式の Ret-Shelter Loader の動作について説明する. Ret-Shelter Loader は実行コードのファイルをストレージから読みだし，自身のメモリ領域を書き換えつつ実行処理を行う. Ret-Shelter Loader での処理は，大きく 3 つの機能から構成されている：

(1) 実行コードのロード時，保護対象の関数を特定
(2) 同じく実行コードのロード時，保護対象個所に ret-shelter モジュールを挿入
(3) 実行時に，ret-shelter がリターンアドレスの退避，関数リターン時の参照の変更，及び，リターンアドレスの検知

以降，詳しく説明する. ただし，(1)と(2)は併せて説明する.

4.2.1 Ret-Shelter Loader の動作

ここで，Ret-Shelter Loader の動作について説明する（図 7 参照）．

(1) 保護対象の実行コードをファイルよりメモリに読み込む．以下は，保護対象の ELF 形式の実行コードに関する処理とする．

(2) ELF32_Ehdr 構造体のメンバ変数を用いて，ELF ヘッダから以下の情報を取得する．
- e_shoff が指す，ELF ファイルの先頭からセクションヘッダテーブルの先頭までのオフセット
- e_shnum が指す，セクションヘッダテーブルに含まれるセクションヘッダの数
- e_shentsize が指す，セクションヘッダテーブルに含まれる各セクションヘッダのサイズ
- e_shstrndx が指す，セクションヘッダテーブルに含まれる shstrtab セクションのインデックス

(3) (2)で取得した情報を用いて，symtab セクションのセクションヘッダを探索する．そして，ELF32_Shdr 構造体のメンバ変数を用いて，symtab セクションのセクションヘッダから以下の情報を取得する．
- sh_offset が指す，ELF ファイルの先頭から symtab セクションの先頭までのオフセット
- sh_size が指す，symtab セクションのサイズ
- sh_entsize が指す，symtab セクションにある各エントリのサイズ

(4) (3)で取得した情報を用いて，symtab セクションを探索し，このセクションにある関数のシンボルを用意した配列に格納する．ただし，リンクする全てのオブジェクトファイルから参照可能なタイプの関数のシンボルだけを対象とする．

(5) ELF32_Ehdr 構造体のメンバ変数を用いて，ELF ヘッダから以下の情報を取得する．
- e_phoff が指す，ELF ファイルの先頭からプログラムヘッダテーブルの先頭までのオフセット
- e_phnum が指す，プログラムヘッダテーブルに含まれるプログラムヘッダの数
- e_phentsize が指す，プログラムヘッダテーブルに含まれる各プログラムヘッダのサイズ
- e_entry が指す，エントリポイントのアドレス

(6) (5)で取得した情報を用いて，プログラムヘッダテーブルから，タイプが LOAD 及び DYNAMIC であるプログラムヘッダを探索する．LOAD タイプのプログラムヘッダの場合，(7)から(12)までの処理を行う．DYNAMIC タイプのプログラムヘッダの場合，(13)から(15)までの処理を行う．いずれの場合も，プログラムヘッダに関する処理を終えた後，(16)の処理を行う．

(7) ELF32_Phdr 構造体のメンバ変数を用いて，プログラムヘッダから以下の情報を取得する．
- p_offset が指す，ELF ファイルの先頭からセグメントの先頭までのオフセット
- p_filesz が指す，セグメントのサイズ
- p_vaddr が指す，セグメントのロード先となる仮想アドレス
- p_flags が指す，セグメントに対する読み込み，書き込み，実行処理のフラグ

(8) memcpy 関数を用いて，セグメントを 1byte 単位で仮想メモリ空間に順次ロードする．この際，(4)で用意した配列を用いて，ロード先のアドレスが関数の始点アドレスかどうかを確認する処理を(9)にて行う．セグメントをロードし終えた後，(6)に戻る．

(9) ロード先のアドレスが，配列に格納されているいずれかの関数のアドレスと一致する場合は，(10)から(12)の処理を行う．一致しない場合は，(8)に戻り，ロード処理を再開する．

(10) ロード先のアドレスを関数の始点アドレスとみなし，ret-shelter モジュールを関数の先頭と終端に挿入する．

(11) (10)にて ret-shelter モジュール（17byte）を挿入することで，本来の命令のロード先アドレスにずれが生じる．そのため，ret-shelter モジュールにより挿入された命令コードの個数を保持しておく．

(12) nop 命令をロードする際には，(11)で保持した個数分の nop 命令を削除する．これにより，ロード先アドレスのずれを補正する．その後，(8)に戻り，ロード処理を再開する．

(13) ELF32_Phdr 構造体のメンバ変数を用いて，プログラムヘッダから以下の情報を取得する．
- p_offset が指す，ELF ファイルの先頭からセグメントの先頭までのオフセット

(14) セグメントを参照し，動的リンクに必要な各セクションの情報を取得する．

(15) (14)で取得したセクションの情報をもとに，動的リンクによってリンクされる関数のシンボルの解決及び再配置処理を行う．その後，(6)に戻る．

(16) (5)で取得したエントリポイントに処理を移すことで，プログラムが実行される．

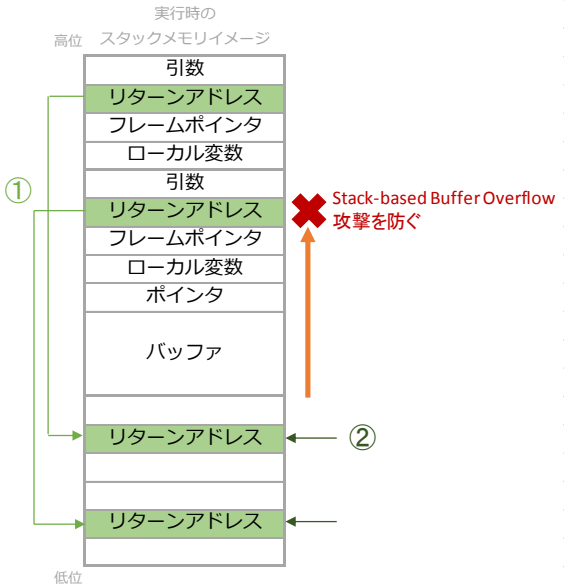


図 8 ret-shelter モジュール適用時のスタックメモリ

イメージ

4.2.2　ret-shelter モジュールの動作

Ret-Shelter Loader にて，保護対象の実行コードをロード時に挿入した ret-shelter モジュールにより，保護対象の実行コードにおける関数Xが関数Yより呼び出される際，通常のスタックの処理の後，リターンアドレスがスタックから下位の（保護対象の実行コードに）利用されないアドレス空間へ退避される（図 8 の①参照）．また，関数 X が終了し，呼び出した関数 Y へリターンする時には，退避したリターンアドレスに実行が移る．その際，リターンアドレスが不正に書き換えられている場合，それを検出する（図 8 の②参照）．

5　評価

5.1　実験環境

以下の実験環境で，Ret-Shelter Loader の動作を確認した．

- Ubuntu 12.04 32bit
- Linux Kernel (3.13.0-40-generic)
- GCC Version 4.6.3

この環境に加えて，Linux Kernel 3 系であれば問題なく動作することが確認できた．

5.2　攻撃の適用と提案方式の動作の確認

本論文では，提案方式である Ret-Shelter Loader の評価のため，Stack-based Buffer Overflow を悪用するいくつかの攻撃を防御することができるかどうかの実験を行った．

今回，実験には，Stack-based Buffer Overflow の脆弱性のあるターゲットプログラム（図 9 参照）を使用した．また，Canary 偽装に対しては，Improper Null Termination の脆弱性のあるターゲットプログラム（図 10 参照）を使用した．

```
void bof(char argv[]){
    char buf[100] = {};
    printf("buf = %p¥n", buf);
    strcpy(buf, argv); //※1
    puts(buf); //※2
    return;
}
```

※1　Stack-based Buffer Overflow 脆弱性がある．ここで，インジェクションベクタを流し込む
※2　Return-to-strcpy 攻撃による GOT 書き換え攻撃を行う際に，puts 関数の.got.plt を書き換える

図 9　Stack-based Buffer Overflow 脆弱性のあるターゲットプログラムの関数

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

int main()
{
    int size;
    char buf[100];
    char line[10];
    setlinebuf(stdout);
    fgets(buf, sizeof(buf), stdin);
    size = atoi(buf);
    fgets(buf, sizeof(buf), stdin);
    strncpy(line, buf, size); //  ※1
    puts(line); //※2
    gets(line); //※3
    return 0;
}
```

※1　Improper Null Termination 脆弱性がある
※2　ここで，canary 値の読み出し
※3　ここに，インジェクションベクタを流し込む

図 10　Improper Null Termination 脆弱性のあるターゲットプログラム

これらのターゲットプログラムを GCC によりコンパイルし，生成された実行コードを，Ret-Shelter Loader を用いて，起動した上で，以下の攻撃を行ったところ，Ret-Shelter Loader の防御機能により，攻撃は成功しなかった．つまり，攻撃を防げた．

- Stack-based Buffer Overflow 攻撃
- return-to-libc 攻撃
- Return-Oriented-Programming 攻撃

また，OS による対策技術 （コード実行防止機能，ASLR，及び，ASCII-armor）が有効な環境において，コンパイラ（Automatic Fortification，及び，RELRO）による対策技術を適用したバイナリに対しても動作した．ただし，SSP に関しては，SSP の本来の目的であるスタックの実行時の書き替えを，Ret-Shelter Loader 自体が行ってしまうので，共存はできない．

5.3　処理時間

Ret-Shelter Loader によって，ターゲットプログラムを起動し実行した際の起動時間を計測した．この時，ターゲットプログラムの起動時間は短いため，5000 回実行した時の合計の処理時間を計測値とした．

Ret-Shelter Loader を利用しない場合，すなわち，OS の ELF Loader を用いて，ターゲットプログラムの起動時間は，9.6885702 秒であった．Ret-Shelter Loader を利用した場合の起動時間は，12.535899 秒であった．提案システムを利用すると起動時間が約 29%増加した．ただし，これは，5000 回実行した時の累積の増加であることに注意されたい．

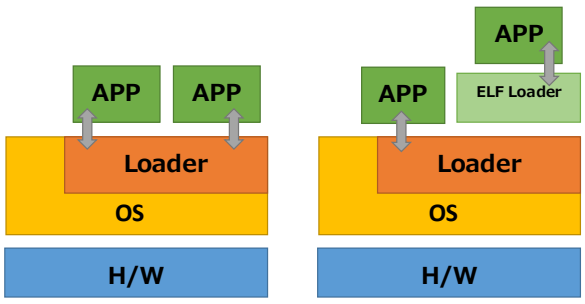
ターゲットプログラムの関数内に，処理にかかる時間を計測するコードを追記し，その関数を 10 万回実行さ

せた合計の処理時間を OS の ELF Loader で実行した場合と Ret-Shelter Loader で実行した場合とで測定し，モジュールの処理時間の割合を算出した．前者の平均は 7.5136496 秒，後者の平均は 7.6082168 秒と，1.26%のオーバーヘッドが計測された．ただし，この場合，ターゲットプログラムの関数の命令数が 65 命令（119 バイト）に対して，ret-shelter モジュールとしての追加の分が，3 命令（17 バイト）と，大きいので，影響がでていると推測される．

6 考察

ここで，既存の対策技術と比較して，Ret-Shelter Loader の優位な点を中心に特徴を概観する．

- 既存の Linux（32bit OS）に対して，一切手を加えずに適用可能 （図 11 参照）
- Linux Kernel 上であれば，OS の種類やバージョンに関係なく動作する
- OS による対策技術（コード実行防止機能, ASLR, 及び，ASCII-armor）と共存できる
- 既存のコンパイラによる対策技術（Automatic Fortification，及び，RELRO）を適用したバイナリに対しても適用できる．ただし，ロード時にリンクしてしまうので，Full-RELRO（遅延バインド無効）の場合であっても，Partial-RELRO（遅延バインド有効）になってしまう．また，現状で SSP とも共存できない
- 実行プログラムの処理時間は，オリジナルの実行コードのそれと比べて，1.29 倍程度になる可能性がある
- 提案方式によって実行ファイルが増加するということはない



(a) 通常のLinuxシステム　(b) 提案方式のLinuxシステム

図 11 通常の Linux システムと提案方式の Linux システムの違い

アプリケーションプログラマが開発時に，独自に作成したプログラムには，脆弱性が入り込んでしまう可能性がある．さらに，また，アプリケーションプログラマの知識不足や開発上の制約で，適切な対策技術を適用せずに，ソフトウエアをリリースしてしまった場合でも，本論文での提案手法である Ret-Shelter Loader は適用することができる．すなわち，本論文で提案する Ret-Shelter Loader は，脆弱性が含まれ，対策技術が適用されていないケースであっても，有効な対策となる可能性があるといえる．

7 まとめ

本論文では，Stack-based Buffer Overflow 攻撃への既存の対策技術が適用されていない，もしくは，適用できないソフトウエアにおいて，Stack-based Buffer Overflow 攻撃を防止する ELF Loader を提案し実装した．

提案方式では，保護対象の実行コードをメモリにロードする際に，提案システムである ELF Loader が，実行コードに対し，動的に対策モジュールを挿入することにより，リターンアドレスの書き換えを困難とする方式を提案した．

提案方式は，プログラムをロードし実行する ELF Loader において実行ファイルに修正を加えるため，コンパイラの既存対策のようにソフトウエアを再コンパイルすることなく対策技術を施すことが可能となる．

ソースコードが取得可能であるソフトウエアに対して，既存の対策技術の多くは効果が高い．しかし，既存の対策技術の多くはコンパイル時の対策がほとんどであるので，ソースコードが取得可能でないソフトウエアに対しては，効果が限定的である．また，提案方式は，既存の対策技術を回避する攻撃に対する対策としても有効であると考えられる．

謝辞：本研究を進めるにあたって，鈴木舞音氏には協力頂いた．記して感謝する．

参考文献

[1] CWE: Common Weakness Enumeration, http://cwe.mitre.org/index.html
[2] NVD: National Vulnerability Database, http://nvd.nist.gov/
[3] マイクロソフトインテリジェンスレポート第16版 (2013年12月), http://www.microsoft.com/ja-JP/download/details.aspx?id=42646
[4] CWE-121: Stack-based Buffer Overflow, http://cwe.mitre.org/data/definitions/121.html
[5] The Linux ELF HOWTO, http://cs.mipt.ru/docs/comp/eng/os/linux/howto/howto_english/elf/elf-howto.html#toc1
[6] CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), http://cwe.mitre.org/data/definitions/120.html
[7] Infosec Writers,“Bypassing non-executable-stack during exploitation using return-to-libc”, http://www.exploit-db.com/download_pdf/17286
[8] Müller, Tilo. "ASLR smack & laugh reference." Seminar on Advanced Exploitation Techniques. 2008.
[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CSS), October 2008.
[10] Linux exploit development part 4 - ASCII armor bypass + return-to-plt, http://www.exploit-db.com/download_pdf/17286

[11] 鈴木舞音，上原崇史，金子洋平，堀洋輔，馬場隆彰，齋藤孝道，メモリ破損脆弱性に対する攻撃の調査と分類，コンピュータセキュリティシンポジウム 2014 論文集 CD-ROM p.767-p.774
[12] Vallentin, Matthias. "On the evolution of buffer overflows." Munich, May (2007).
[13] Röttger, Stephen. "Malicious Code Execution Prevention through Function Pointer Protection." (2013).
[14] オープンソース・ソフトウェアのセキュリティ確保に関する調査報告書，第Ⅲ部，セキュアな実行コードの生成・実行環境技術に関する調査，https://www.ipa.go.jp/files/000013695.pdf
[15] T. Bletch, X. Jiang, V. X. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack”, In Proceedings of the 6th ACM Conference on Computer and Communications Security (CSS), 2011, pp. 30-40
[16] Payer, Mathias, and Thomas R. Gross. "String oriented programming: when ASLR is not enough." Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. ACM, 2013.
[17] 齋藤孝道，鈴木舞音，上原崇史，金子洋平，角田佳史，馬場隆彰，メモリ破損攻撃への対策技術の調査と分類，コンピュータセキュリティシンポジウム 2014 論文集 CD-ROM p.775-p.782
[18] 齋藤孝道,マスタリング TCP/IP 情報セキュリティ編,オーム社(2013)
[19] "Exec Shield", new Linux security feature, http://lwn.net/Articles/31032/
[20] A stack smashing technique protection tool for Linux, http://www.angelfire.com/sk/stackshield/
[21] Stack Shield A "stack smashing" technique protection tool for Linux, http://www.angelfire.com/sk/stackshield/download.html

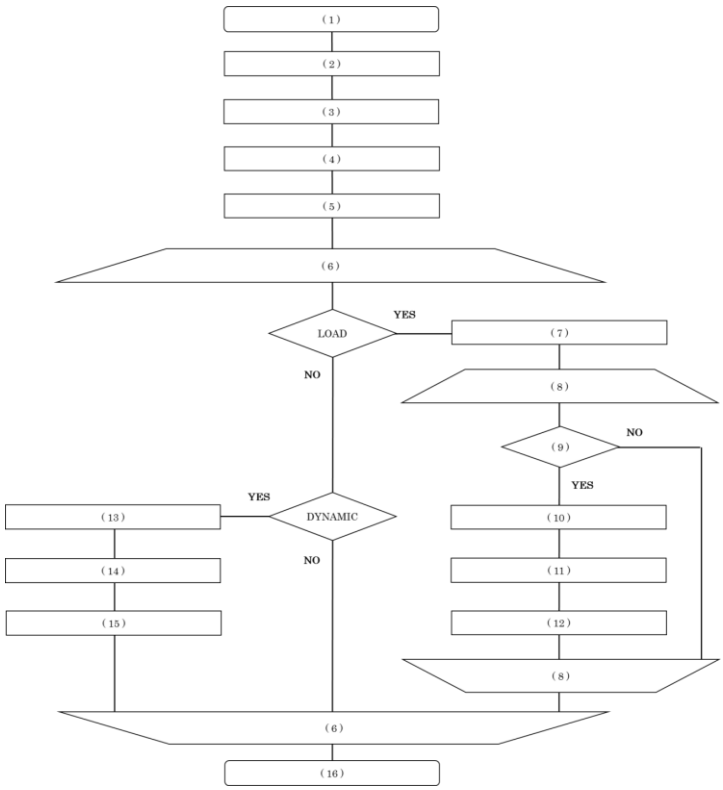


図 7　Ret-Shelter Loader の動作