# Linux上のメモリ破壊攻撃群に対応するプログラムローダ

渡辺 亮平[1]　近藤 秀太[1]　菅原 捷汰[1]　横山 雅展[1]　中村 慈愛[2]　須崎 有康[3]　齋藤 孝道[2]

**概要**：実行バイナリにおける脆弱性の一つであるメモリ破壊脆弱性は，現在でも報告が絶えない．一方で，これまでに，コンパイラ，リンカ，OS，ライブラリにおける対策技術が提案・実装されてきた．しかし，コンパイラやリンカにおける対策技術は，ソースコードが必要で，主に，開発フェーズでの適用を想定する．また，ライブラリにおける対策技術は，バージョンなどの依存関係により適用できない場合がある．加えて，我々の先行研究により，主要なLinuxディストリビューションにおいて，コンパイラのセキュリティオプションが適用されていないバイナリが一定数存在することが示された．本論文の調査で，新たに「メモリ破壊を招くライブラリ関数」が現在でも一定数利用されていることがわかった．そこで，本論文では，ソースコードが入手できない配布済みの実行バイナリへ適用できるアプリケーションレベルのプログラムローダを用いたメモリ破壊攻撃への総合的な対策を提案する．提案手法では，「メモリ破壊を招くライブラリ関数」をより安全な代替関数に置換をすることで，代表的なメモリ破壊攻撃を緩和する．

**キーワード**：メモリ破壊脆弱性，メモリ破壊攻撃

# Safe Trans Loader: Mitigation and Prevention of Memory Corruption on Linux

Ryohei Watanabe[1]　Shuta Kondo[1]　Shota Sugawara[1]　Masahiro Yokoyama[1]　Jiai Nakamura[2]
Kuniyasu Suzaki[3]　Takamichi Saito[2]

***Abstract:*** Memory corruption vulnerabilities continue to be serious threats and reported even now. Until today, many countermeasures in compiler, linker, OS, and library have been proposed and implemented. However, countermeasures in compilers and linkers require source code, and are expected to be mainly applied in the development phase. Also, library countermeasures may not be applicable depending on its dependency version. Moreover, our previous research showed that there were a certain number of binaries without compiler security options in major Linux distributions. We found newly that a certain number of "library functions causing memory corruption" such as strcpy function is still being used in the binaries. In this paper, we propose comprehensive countermeasure against memory corruption attacks by using application-level program loader which can apply to distributed binaries. Our proposed method mitigates representative memory corruption attacks by replacing "library functions causing memory corruption" with safer alternative functions.

***Keywords:*** Memory Corruption Vulnerability, Memory Corruption Attack

## 1. はじめに

CWE-119[1] に分類されるメモリ破壊脆弱性は現在でも報告が絶えない脆弱性の一つである．中でも，CWE-121[2] に分類される Stack-based Buffer Overflow（以降，SBoF と呼ぶ）脆弱性，CWE-122[3] に分類される Heap-based Buffer Overflow（以降，HBoF と呼ぶ）脆弱性の報告件数は現在でも一定数を保っている．また，CWE-416[4] に分類される Use After Free 脆弱性は，2006 年に登場後，報告件数は増加傾向にある．

これらの脆弱性を悪用するメモリ破壊攻撃は，システ

[1] 明治大学大学院
Graduate School of Meiji University
[2] 明治大学
Meiji University
[3] 国立研究開発法人産業技術総合研究所
National Institute of Advanced Industrial Science and Technology

ムのクラッシュや端末制御の奪取を招く可能性がある．これまでに，それらの攻撃に対して，コンパイラ，リンカ，OS，ライブラリにおける対策技術が提案されてきた．一部の対策技術は主要な OS やコンパイラに組み込まれ，ASLR[5] や DEP[6]，SSP[7] といった代表的なセキュリティ機構はデフォルトで有効となっている．しかし，それらの対策技術は，コード再利用攻撃 [8][9] を始めとする様々な手法によって回避されることが知られている．近年の研究では，コード再利用攻撃を困難にする Software Diversity[10][11][12]，プログラムの制御フローを検査する Control-flow integrity [13][14][15]，境界外アクセスを検知する Bounds Checking[16][17][18][19][20] が提案されている．

しかし，既存の対策技術は効果がある一方で，状況によっては利用できない場合がある．例えば，コンパイラやリンカにおける対策技術は開発フェーズでの適用が想定され，その利用には，ソースコードの取得と再コンパイルを必要とする．また，OS における対策技術は，システム全体へ影響を及ぼすため，新しい OS に切り替えることが困難な環境などではその恩恵を享受できない．ライブラリによる対策は，運用フェーズでの適用が可能であるが，バージョンなどの依存関係がない場合に限られる．すなわち，脆弱性が発見される運用フェーズにおいて，既存対策は利用できないケースがあることがわかる．

加えて，我々の先行研究により，主要な Linux ディストリビューションにおいて，コンパイラのセキュリティ対策が機能していないバイナリが一定数存在することが示された [23]．さらに，こうしたバイナリの中に「メモリ破壊を招くライブラリ関数」を現在も利用しているものが，一定数存在することが本論文の調査で新たにわかった．メモリ破壊を招くライブラリ関数に起因する脆弱性は，CVE-2017-14492[21]，CVE-2017-14493[22] として現在でも報告されており，当該脆弱性対策が期待される Automatic Fortification の適用によっても検知することができないケースも存在する．

以上より，本論文では，実行バイナリへ適用を可能とするアプリケーションレベルのプログラムローダ（以降，Safe Trans ローダと呼ぶ）を用いたメモリ破壊攻撃への総合的な対策を提案する．Safe Trans ローダは，実行時にメモリ破壊を招くライブラリ関数を含む，複数のライブラリ関数をより安全な代替関数に置き換えることで，その関数における SBoF 攻撃，HBoF 攻撃および，Use After Free 攻撃を緩和する．また，対象ソフトウエアがリリースされた後，運用フェーズで脆弱性を対処する際，ソースコードが入手できないことが想定される．そのようなケースにおいても対策を適用できることが特徴である．

本論文での実装においては，32 ビット Linux OS における ELF ファイルへの適用を想定して Safe Trans ローダのプロトタイプを実装した．Safe Trans ローダを，いくつかのバイナリに適用したところ，SBoF 攻撃，HBoF 攻撃および，Use After Free 攻撃の緩和を確認できた．また，SEPC CPU2006 を用いて評価を行った結果，オーバーヘッドは，0.64%であることがわかった．

## 2. 背景

本論文では，新たにメモリ破壊を招くライブラリ関数の使用状況を調査した．本章では，2.1 節で先行研究の結果について，2.2 節で新たに行った調査の結果について述べる．

### 2.1 コンパイラのセキュリティオプションの適用状況

先行研究では，3 つのディストリビューション（CentOS，OpenSUSE，Ubuntu）の 3 世代において，4 つの対策技術（SSP，Automatic Fortification，RELRO，PIE）の適用状況の調査を行なった [23]．これらの対策技術は，ソースコードのコンパイル時にセキュリティオプションを指定することで適用される．調査の結果，それぞれのディストリビューションにおいて，世代が上がるにつれて適用率が高くなる傾向があることがわかった．

比較的新しいディストリビューションにおける各対策技術の適用率を**表 1** に示す．PIE を除いた 3 つの対策技術は半数以上の実行バイナリで対策技術が適用されていたが，この調査によって適用されていない実行バイナリが一定数存在することが明らかとなった．

どのディストリビューションにおいても PIE の適用率は低かった．これら対策技術が適用されていないバイナリの中には，オーバーヘッドの問題やセキュリティ機構によって正しく動作しなくなる問題のため，明示的に無効にしているケースがあることがわかった．

ビルド時にセキュリティオプションを有効にしても，対策技術の適用条件を満たしておらず，セキュリティ機構が組み込まれないケースも 3 つの対策技術のいずれにでも散見された．これらのケースでは，開発者がセキュリティオプションを正しく理解していない場合に，望んだセキュリティ機構が適用されず結果として無駄なビルドを招く可能性がある．

Ubuntu のある実行バイナリでは，SSP の適用条件 [24] を満たし，かつ SSP のセキュリティオプションが有効にされているにも関わらず，実際にビルドされた実行バイナリには SSP のセキュリティ機構が組み込まれていなかった．これは，バグの可能性も考えられる．

### 2.2 メモリ破壊を招くライブラリ関数の使用状況

strcpy 関数や gets 関数のようなメモリ破壊を招くライブラリ関数の利用は古くから問題視されており，文献 [25] や [26] ではこれら関数の利用を推奨していない．しかし，過去の調査過程で，これらライブラリ関数を利用する実行

表 1　主要なディストリビューションにおける対策技術の適用率

| 対策技術 | CentOS7.3 | openSUSE13.2 | Ubuntu14.04 |
|---|---|---|---|
| SSP | 91% | 65% | 74% |
| Automatic Fortification | 85% | 65% | 74% |
| PIE | 26% | 11% | 15% |
| Partial RELRO | 75% | 97% | 83% |
| Full RELRO | 25% | 3% | 13% |

バイナリが散見された．そこで，我々は，新たにメモリ破壊を招くライブラリ関数の利用状況の調査を行った．

#### 2.2.1　調査対象

先行研究 [23] に習い，同様のディストリビューション対してメモリ破壊を招くライブラリ関数の使用状況の調査を行なった．

ここで，「メモリ破壊を招くライブラリ関数」とは，glibc-2.25 における Automatic Fortification[27] が置換対象とする計 79 個とした．

調査対象の実行バイナリは，ディストリビューションに含まれるデフォルトで PATH が通っているディレクトリ内の実行バイナリとした．調査対象を**表 2** に示す．

表 2　調査対象のディストリビューションおよび実行バイナリ

| ディストリビューション | 実行バイナリの総数 |
|---|---|
| CentOS7.3 | 1,601 |
| openSUSE13.2 | 2,207 |
| Ubuntu14.04 | 1,159 |

#### 2.2.2　調査方法

調査対象の実行バイナリにおけるシンボル情報からメモリ破壊を招くライブラリ関数の使用の有無を判別する．なお，Automatic Fortification によって置換された_chk を接尾辞に持つ関数は，境界検査を行うのでメモリ破壊を招くライブラリ関数には含まれない．glibc-2.25 から Automatic Fortification が置換対象とする関数の取得は文献 [28] を参考にした．

#### 2.2.3　調査結果

各ディストリビューションにおけるメモリ破壊を招くライブラリ関数の使用状況を**図 1** に示す．

結果から，どのディストリビューションにおいても，75%以上の実行バイナリにおいて，メモリ破壊を招くライブラリ関数が利用されていることがわかる．また，図 1 には，メモリ破壊を招くライブラリ関数を利用している内，Automatic Fortification が適用されているバイナリを示している．結果から，Automatic Fortification が適用されている場合においても，対象関数が全て置換されず，多くの実行バイナリでメモリ破壊を招くライブラリ関数を使用していることがわかる．

Automatic Fortification は，コンパイル時に以下の条件を満たす場合，関数の置換を行わない [26].

(1) 対象関数において，メモリ破壊が起きないことが自明な場合

(2) 対象関数において，書き込み先バッファのサイズを判断出来ない場合

これら実行バイナリが，(1) に該当する場合，安全な関数呼び出しであると言えるが，(2) に該当する場合，その関数に起因するメモリ破壊攻撃を検知することができない．実際に，我々の環境では，CVE-2009-2957[29] や CVE-2017-14493[21] に対して Automatic Fortification を有効にしてコンパイルを行ったが，関数の置換が行われず，攻撃を防ぐことができなかった．

以上から，Automatic Fortification の適用の有無に関わらず，メモリ破壊を招くライブラリ関数の利用は，その実行バイナリにおける潜在的な脅威に成りうる．

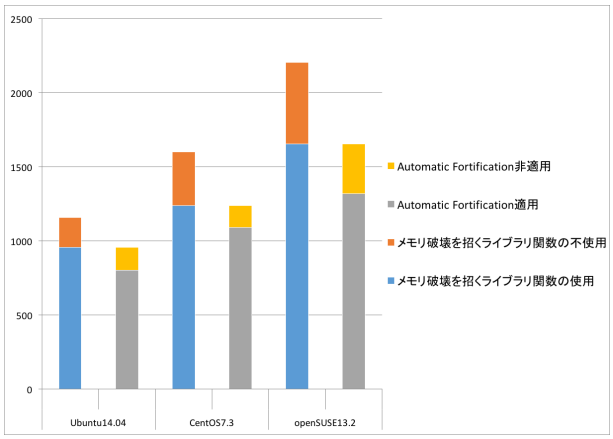


図 1　メモリ破壊を招くライブラリ関数の使用状況

## 3.　既存のメモリ破壊攻撃への対策技術

メモリ破壊攻撃への対策技術は様々に考案され，実装されてきた．しかし，既に配布された実行バイナリに適用でき，かつ種々のメモリ破壊攻撃を包括的に緩和できる効果的な対策技術は少ない．Szekeres らによると，広く採用されうる対策技術のオーバーヘッドは、5%から 10%以下でなければならない [30]．本節では，メモリ破壊攻撃への様々な対策技術を取り上げる．

### 3.1　Buffer Overflow 攻撃への対策技術

古くから知られている対策技術に Stack Smashing Protector[7] がある．これはカナリアと呼ばれる値をフレームポインタとローカル変数の間に挿入し，その値が書き換えられた時に SBoF 攻撃を検出する．Bounds Checking[16][17][18][19][20] は生成されたオブジェクトのサイズやアドレスを生成時に保存しておき，利用される際に，オブジェクトのサイズをチェックすることで，Buffer Overflow 攻撃を検出する．しかし，これらの対策技術は適用にソースコードが必要なので，すでに配布されたバイナリに対して適用できない．

Libsafe[31] は SBoF 脆弱性を招くライブラリ関数を，境界検査処理をする代替関数に置換することで SBoF 攻撃を緩和する．しかし、Libsafe はすでに開発が終了しており，保守されていない．さらに，setuid が設定されたバイナリに対して適用する場合，環境変数 LD_PRELOAD[32] が無視されてしまうので関数の置換ができない問題がある．この問題の回避策として Libsafe は，システム全体への適用を提供しているが，この場合，システム上で動作する全てのバイナリに対して関数の置換が行われてしまうので，バイナリごとに適用，非適用を選択できない．

StackArmor[33] は実行前に静的にバイナリを書き換え，実行時にスタック上のデータの配置をランダムにすることで SBoF 攻撃を緩和するが，オーバーヘッドが大きいという問題がある．

### 3.2 Use After Free 攻撃への対策技術

CETS[34] はメモリ領域の確保時にその領域の参照情報を管理する領域を確保し，ポインタによるメモリ領域へのアクセス時に参照情報を確認することで実行時にダングリングポインタを検知する．

DANGNULL[35]，FreeSentry[36] はポインタの参照情報を管理し，ポインタ操作のたびに参照情報を更新していく．ポインタ操作の結果，ダングリングポインタとなる場合は null を代入することでダングリングポインタの発生を防ぐ．しかし，これらの対策技術は適用にソースコードが必要なので，すでに配布されたバイナリに対して適用できない．

Cling[37] は解放済みのメモリ領域の再利用を特定の条件に当てはまる場合に限定することでダングリングポインタを悪用した攻撃を緩和する．

Dieharder[38] はヒープ上のランダムな位置からメモリ確保を行うことでダングリングポインタを悪用した攻撃を緩和する．

総じて，既存の Use After Free 攻撃への対策技術にはオーバーヘッドが大きいという問題がある．

### 3.3 その他の対策技術

上記の攻撃以外のメモリ破壊攻撃への静的な対策技術には，テイント解析技術 [39][40]，アドレスランダマイズ化 [10][11][12]，Code-pointer integrity[41]， Control-flow integrity（CFI）[13][14][15] がある．

ASLR[5] は実行時にスタック，ヒープ，共有ライブラリをランダマイズ化することでメモリ破壊攻撃を緩和するが，攻撃自体を防ぐことはできない．CFI は間接分岐およびリターンアドレスの正当性を検証することで，不正な命令の実行を防ぐが，メモリ破壊自体を防ぐことはできない．

## 4. 提案方式

### 4.1 脅威モデル

Safe Trans ローダの適用による有効性を示すために，本論文では，SBoF，HBoF，Use After Free の計 3 つの攻撃を脅威モデルと想定する．SBoF および HBoF は，2 章の調査結果で示したメモリ破壊を招くライブラリ関数に起因する攻撃を対象とし，Use After Free は，それぞれの脆弱性を悪用した攻撃を対象とする．

### 4.2 Safe Trans ローダの設計

本論文では，メモリ破壊攻撃を緩和する Safe Trans ローダを提案する．**図 2** に Safe Trans ローダ適用の概念図を示す．Safe Trans ローダは，アプリケーションレベルで動作し，OS 標準のローダに代わって，保護対象バイナリをロードする．その際に，保護対象バイナリ内に Safe Trans ローダが置換対象とする関数（以降，置換対象の関数と呼ぶ）が見つかった場合は，それぞれの攻撃への対策処理を加えた代替関数への置換を行う．Safe Trans ローダは，ローダというアプローチを採用しているので，ソースコードの再コンパイルをせずにバイナリに対して直接適用できることが特徴である．

ここで，本論文における置換対象の関数一覧を**表 A·1** に定める．これら関数は，Automatic Fortification が置換する関数および文献 [25] を参考に我々が選定した．

SBoF および HBoF 攻撃への対策では，置換対象の関数のうち，SBoF と HBoF に関連するものを，境界検査処理を追加したより安全な関数（以降、SBoF/HBoF 境界検査関数と呼ぶ）に置換する．HBoF の対策では，メモリ破壊を招くライブラリ関数だけでなく，境界検査を行うための関連するメモリ確保関数およびメモリ解放関数も代替関数に置換する．Use After Free 攻撃への対策では，既存のメモリ解放関数をメモリブロックの解放処理を遅延させる処理を追加した関数に置換する．

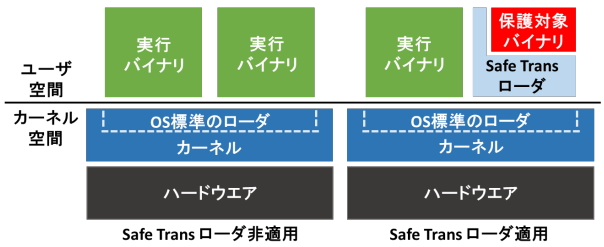


**図 2** Safe Trans ローダ適用の概念図

### 4.3 Safe Trans ローダの動作フロー

Safe Trans ローダは，実行時に保護対象バイナリのパスを受け取る．OS 標準のローダによって仮想メモリにロー

ドされた Safe Trans ローダは，以下の順序で保護対象バイナリをロードする．

(1) 保護対象バイナリの読み込み
この後の処理で必要となる保護対象バイナリの情報を取得するために，Safe Trans ローダは，保護対象バイナリを Safe Trans ローダ自身のヒープ領域に読み込む．

(2) 保護対象バイナリの ELF ヘッダの解析
保護対象バイナリをヒープ領域に読み込んだ後，Safe Trans ローダは，保護対象バイナリの ELF ヘッダを解析し，そのロードに必要な情報を取得する．

(3) 保護対象バイナリのマッピング処理
Safe Trans ローダは，2 で取得したオフセットから保護対象バイナリのプログラムヘッダを走査する．プログラムヘッダの走査中に LOAD セグメントに対応するものがあれば，Safe Trans ローダはそのプログラムヘッダに記述されている情報に従い，LOAD セグメントを Safe Trans ローダ自身の仮想メモリにマップする．保護対象バイナリの動保護対象バイナリのプログラムヘッダの走査中に DYNAMIC セグメントに対応するものがあれば，その情報に従い，共有ライブラリをロードし，その後再配置処理を行う．このとき，ロードする共有ライブラリが Safe Trans ローダの実行によりすでにロードされていた場合，共有ライブラリのロードは行われず，ロード済みのものを Safe Trans ローダと共有する．関数の置換は，保護対象バイナリの再配置処理で行われる．

(4) Shadow Memory の作成
HBoF 攻撃への対策において，ヒープ領域内のバッファの境界検査を行うためには，バッファのサイズ情報が必要となる．Safe Trans ローダは，サイズ情報を保持しておく領域として Shadow Memory を作成する．

(5) 保護対象バイナリの実行開始
Safe Trans ローダは，1 で保護対象バイナリを読み込んだヒープ領域を解放し，2 で取得したエントリポイントに制御を移す．

**図 3** に 1 から 6 までの動作が行われたときの仮想メモリの全体像を示す．

### 4.4 関数の置換

一般に，動的リンクを必要とする ELF バイナリのライブラリ関数呼び出しは，.plt セクションを経由し.got.plt（または.got）セクションの呼び出す関数に対応するエントリを参照することで行われる．Safe Trans ローダによる関数置換は，上記の仕組みを利用し，保護対象バイナリの再配置時に，置換対象関数の.got セクションのエントリに，置換後の代替関数のアドレスを書き込むことで行われる．

例として，保護対象バイナリの実行中に，strcpy 関数を，

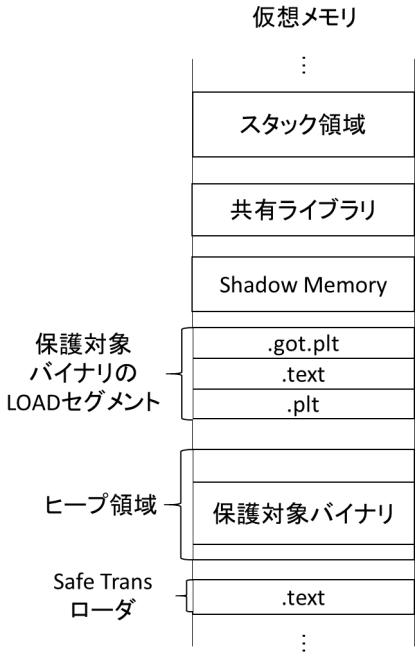


**図 3** 保護対象バイナリ実行時の仮想メモリの全体像

対策処理を追加した Hstrcpy 関数に置換した場合の関数の呼び出しの様子を**図 4** に示す．

保護対象バイナリの.text セクションにおいて strcpy 関数が呼び出されると，.plt セクション，.got.plt セクションの順にセクションを参照し（図 4 の矢印 1，矢印 2），strcpy 関数のアドレスを取得する．.got.plt（または.got）セクションには通常であれば，共有ライブラリ中の strcpy 関数のアドレスが書き込まれるが，Safe Trans ローダがこの部分を Safe Trans ローダの Hstrcpy 関数のアドレスに置換しているので，strcpy 関数の代わりに Hstrcpy 関数が呼び出される（図 4 の矢印 3）．

本提案方式は，上記の動的リンクの仕組みを利用しているので，置換対象関数が実行バイナリと静的リンクされている場合，対策を講じることができない問題がある．

### 4.5 代替関数における対策処理

SBoF 攻撃，HBoF 攻撃，UAF 攻撃のそれぞれに対して，Safe Trans ローダ上に定義した代替関数が行う対策処理について説明する．

#### 4.5.1 SBoF 攻撃への対策

SBoF 攻撃への対策として，フレームポインタを用いたスタック領域内の書き込み先のバッファの境界検査を行う．SBoF 境界検査関数が呼び出されると，はじめに，スタックフレーム内に存在するフレームポインタ辿り，書き込み先のバッファが存在するスタックフレームを特定する．次に，関数内での書き込み先のバッファの上限サイズを求める．この時のバッファの上限サイズは，関数への引数として渡されたバッファのアドレスからバッファと同じスタッ

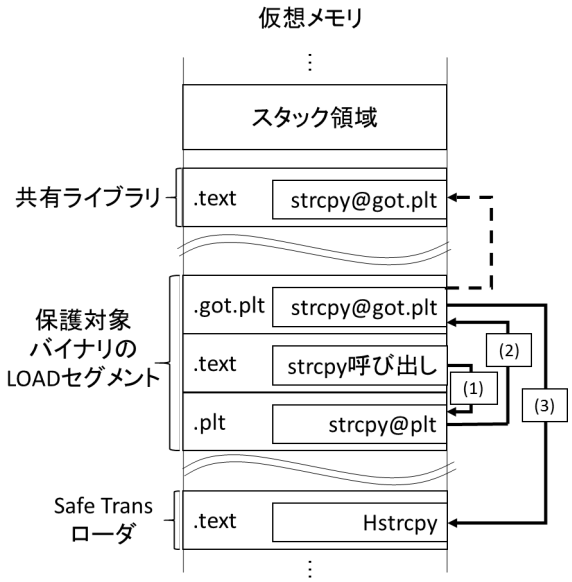


図 4　strcpy 関数を Hstrcpy 関数に置換した場合の関数の呼び出し

クフレーム内のフレームポインタの位置までとする．その後，書き込む文字列のサイズと計算した上限サイズを比較し，書き込む文字列のサイズが上限サイズを超えない場合，書き込み処理を行う．上限サイズを超えた場合，保護対象バイナリの実行を停止する．このようにすることで，SBoF 攻撃を緩和する関数に上限サイズを上回るサイズの文字列が与えられた場合でも，**図 5** に示すようにフレームポインタ以降の書き換えを防ぐ．

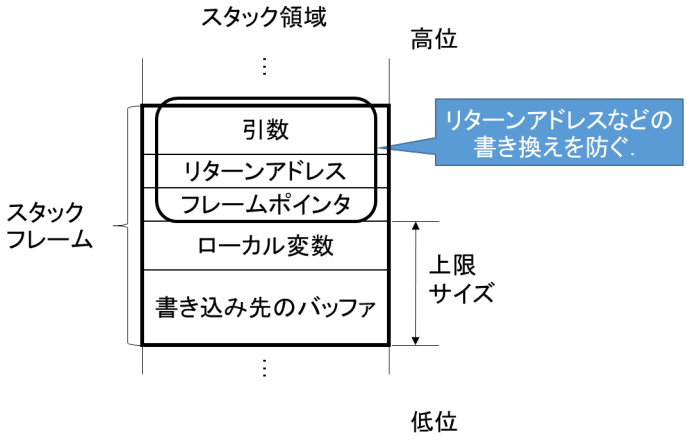


図 5　スタック領域内にある書き込み先のバッファの上限サイズ

### 4.5.2　HBoF 攻撃への対策

HBoF 攻撃への対策として，Address Sanitizer[42] などで採用されている Shadow Memory 技術を用いて，書き込み先のバッファの境界検査を行う．本提案方式において，Shadow Memory は，Safe Trans ローダの起動時に確保され，置換された malloc 関数などの動的メモリ確保関数により，ヒープ領域に確保されたメモリブロックの情報が格納される．HBoF 境界検査関数が呼び出されると，はじめに，引数として渡された書き込み先のバッファアドレスから対応する Shadow Memory のアドレスを計算し，Shadow Memory からメモリブロックのサイズを取得する．次に，書き込む文字列のサイズと Shadow Memory から取得したサイズを比較し，書き込む文字列のサイズが，取得したメモリブロックのサイズを超えない場合，書き込み処理を行う（**図 6**）．メモリブロックのサイズを超えた場合，保護対象バイナリの実行を停止する．置換されたメモリ開放関数では，引数として渡されたポインタから，対応する Shadow Memory のサイズ情報をクリアする．

AddressSanitizer[42] では，スタック領域やヒープ領域などにのメモリ破壊脆弱性を検出するために，データ領域の 8 バイトと，Shadow Memory の 1 バイトを対応付けている．これにより，32bit 環境では，約 512M バイトの領域を確保する必要がある．本提案方式では，メモリ確保関数におけるサイズ情報のみを格納するため，ヒープ領域の 8 バイトと Shadow Memory の 1 ビットを対応付けた．よって，Safe Trans ローダは起動時に，約 48M バイトの領域を確保する．

Shadow Memory を用いたメモリ確保関数およびメモリ開放関数により，Double Free 攻撃への対策も行うことができる．メモリブロックの解放処理の際に，解放するメモリブロックのサイズ情報が Shadow Memory に存在しない場合，そのメモリブロックに対して二重解放が行われていると判断し，プログラムの実行を停止する．

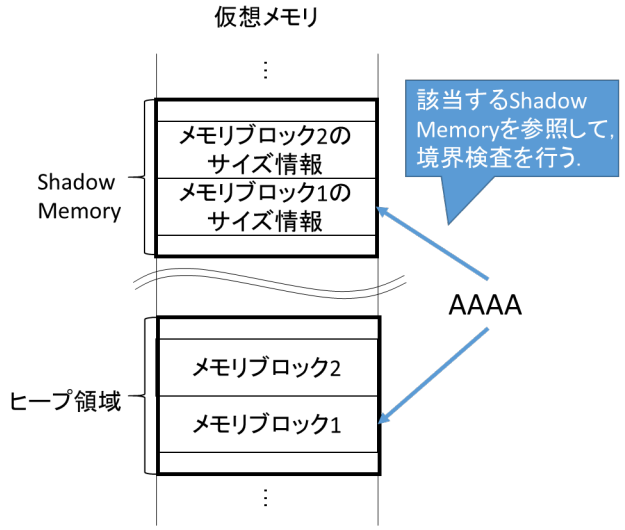


図 6　Shadow Memory を用いた境界検査

### 4.5.3　Use After Free 攻撃への対策

Use After Free 攻撃は，解放直後のメモリブロックが再利用される性質を悪用するケースが多いということが文献 [43] でわかっている．そこで，Safe Trans ローダでは保護対象バイナリで使用されている free 関数を，メモリ解放処理を遅延させる free 関数（以降，Hfree 関数という）に

置換する．

図 7 に Hfree 関数のフローチャートを示す．Hfree 関数が呼び出されると，はじめに，引数で指定したポインタを Safe Trans ローダ上で定義されているキューに保存する．キューは解放予定のメモリブロックを指すポインタを管理する．このとき，保存後のキューに空きがある場合は，メモリブロックの解放処理は行わないが，空きがなくなった場合は，一番古いものを解放する．

メモリ解放処理を遅延させることで，解放直後のメモリブロックを再利用されることがないので，Use After Free 攻撃を緩和できる．

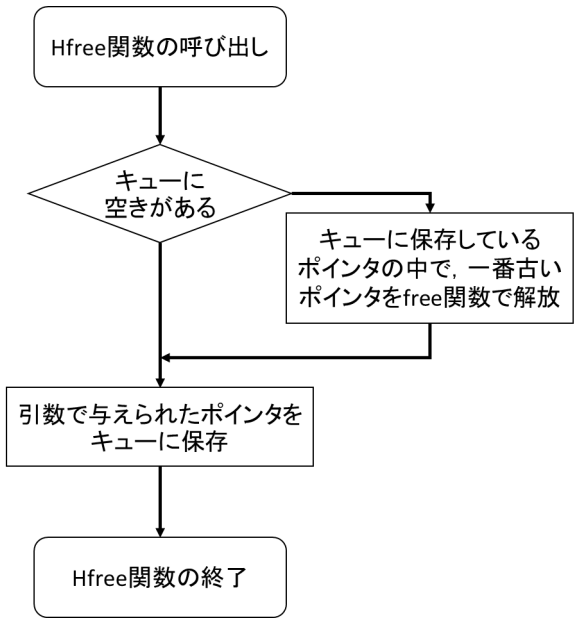


図 7　Hfree 関数のフローチャート

## 5. 評価

本節では，Safe Trans ローダの評価を行う．

### 5.1 評価環境

評価環境として，Intel(R) Xeon(R) CPU E5620@2.40GHz において，Ubuntu14.04 LTS 32bit を用いた．また，コンパイラは gcc4.8.4 を使用した．

### 5.2 メモリ破壊攻撃への有効性

#### 5.2.1 SBoF 攻撃への有効性

Safe Trans ローダによる SBoF 攻撃対策の有効性の確認ため，合計 4 種類のバイナリを用いて提案方式の有効性の評価を行った．

まず，CWE-121 のページ [2] で公開されているサンプルコード (Example1 および Example2) を元に作成した 2 種類のバイナリを用意した．

さらに，実システムへの有効性確認ため，CVE-2013-4256[44] として脆弱性が報告されている Network Audio System1.9.3，CVE-2017-14492[21] として脆弱性が報告されている dnsmasq2.70 を用意した．

Example1 および Example2 は strcpy 関数に起因する脆弱性である．CVE-2013-4256 は strcat 関数に起因する脆弱性である．CVE-2017-14492 は，memcpy 関数に起因する脆弱性である．

これら 4 種類のバイナリを Safe Trans ローダ上で実行し，書き込み先のバッファの境界を超えてリターンアドレスを上書きするサイズの入力を与えた場合に，その書き換えを防ぐことができるかを検証した．その結果，それぞれのバイナリの実行において，Safe Trans ローダはリターンアドレスの書き換えを防ぐことを確認した．

#### 5.2.2 HBoF 攻撃への有効性

Safe Trans ローダによる HBoF 攻撃対策の有効性の確認ため，合計 4 種類のバイナリを用いて提案方式の有効性の評価を行った．

まず，CWE-122 のページ [3] で公開されているサンプルコード (Example1 および Example2) を元に作成した 2 種類のバイナリ (Example1 および Example2) を用意した．

さらに，実システムへの有効性確認ため，CVE-2009-2957[29] として脆弱性が報告されている dnsmasq-2.49，CVE-2017-14492[21] として脆弱性が報告されている dnsmasq-2.70 を用意した．

Example1 および Example2 は strcpy に起因する脆弱性である．CVE-2009-2957 は，strncat 関数に起因する脆弱性である．CVE-2017-14492 は，sprintf 関数に起因する脆弱性である．

これら 4 種類のバイナリを Safe Trans ローダ上で実行し，書き込み先のバッファの境界を超える入力を与えた場合に，その書き換えを防ぐことができるかを検証した．その結果，それぞれのバイナリの実行において Safe Trans ローダはバッファの境界外への書き換えを防ぐことを確認した．

#### 5.2.3 Use After Free 攻撃への有効性

Safe Trans ローダによる Use After Free 攻撃対策の有効性の確認ため，以下のバイナリを用いて提案方式の有効性の評価を行った．

CWE-416 のページ [4] で公開されているサンプルコードを元に作成したバイナリ Example1 を用いて Use After Free 攻撃への有効性の評価を行った．Example1 では malloc 関数によりメモリブロックが確保され，そのメモリブロックが free 関数により解放された直後に，再度 malloc 関数でメモリブロックが確保されることで Use After Free 攻撃が引き起こされる．

Example1 のバイナリを Safe Trans ローダ上で実行し，Use After Free 攻撃を検知することができるかを検証した．その結果，解放されたメモリブロックの再利用が行われなかった．更に，strncpy 関数において，第 1 引数のポ

インタとメモリブロック解放後に動的に確保されたメモリブロックの先頭アドレスが一致しなかった．以上により，Use After Free 攻撃を防ぐことができた．

### 5.3 実行時のオーバーヘッド

Safe Trans ローダの適用が，プログラムの実行速度面にどの程度影響を及ぼすかを調べるために，SPEC CPU2006 ベンチマークが提供する 11 個のバイナリについて，各バイナリを Safe Trans ローダ上で実行した場合と，Safe Trans ローダを用いずに実行した場合の実行時間を測定した．測定結果のグラフを**図 8** に示す．Safe Trans ローダのオーバーヘッドは，平均して約 0.64%であり，非常に小さい．これら実行バイナリ中で，h264ref では，strncpy 関数，strncat 関数，memcpy 関数，snprintf 関数で置換が発生しており，結果として他のものより実行時間が増えたと考えられる．

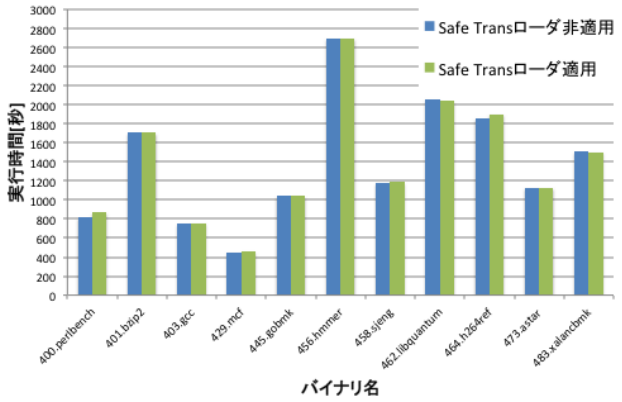


図 8 SPEC CPU2006 の実行結果

## 6. 考察

### 6.1 ソフトウエアライフサイクルの観点からの考察

一般的に，対策技術の適用タイミングについて，ソフトウエアの開発（実装）フェーズと運用フェーズの 2 つの視点で考えることができる．

開発フェーズにおいて適用する対策技術としては，コーディング時に，安全なライブラリを使用する手法 [45] やソースコード解析ツールを用いて脆弱性を作りこまないようにする手法 [46] があるが，万全とは行かない．また，コンパイル・静的リンク時に対策技術を適用する手法が代表的な対策であるが，ソースコードを必要とする．さらに，ソフトウエアの開発者が細心の注意をはらったからといって，配布するバイナリに脆弱性を作り込んでしまう可能性がゼロとはいいきれず，運用フェーズで，脆弱性が発見されるケースも少なくない．また，2 章で示したように，コンパイラにおける対策技術が適用されていない実行バイナリや，適用された実行バイナリ中にも依然として脅威が存在することがある．したがって，開発フェーズでの対策には，限界があるといえる．

一方，運用フェーズで適用する対策には，開発フェーズと完全に切り離して，セキュリティ対策を適用できることが最大のメリットがある．一般的には，ソフトウエアの起動時または動的リンク時に対策技術を適用する手法 [32] がある．運用フェーズに適用する対策技術の特徴は，適用者がソフトウエアの利用者や運用者であることがあり，開発者ほど当該ソフトウエアの知識を持たない可能性があることと，ソースコードを得られない場合がある．しかし，運用フェーズで適用する対策技術の利点は，脆弱性が作り込まれた状態で配布された，もしくは，配布された後，脆弱性が発見されたソフトウエアに対して，利用者側で対策を適用できることである．特に，Safe Trans ローダは，個別のアプリケーションソフトウエアごとに対策を適用できるで，OS の切り替えやライブラリの交換などの，システム全体に及ぼす影響が少ないと言える．さらに，システムサポートが終了しているなどでパッチが提供されないケースや，パッチを適用すると障害が発生するケースにおいては，Safe Trans ローダがその解決策の一つとなりうる．

### 6.2 メモリ破壊攻撃への既存の対策技術との比較

表 3 のようにメモリ破壊攻撃への包括的な対策技術は少ない．一方で，Safe Trans ローダはメモリ破壊攻撃への対策を 1%未満のオーバーヘッドで可能にしている．個々の攻撃への対策技術では，Safe Trans ローダよりも厳密なメモリ検査をする既存の対策技術もある．しかし，それらは既に配布されたバイナリに対して適用できない問題や，オーバーヘッドの問題がある．5 章（評価）で示した通り，Safe Trans ローダは，脆弱性が報告されている実バイナリにおける様々なメモリ破壊攻撃を防ぐことができる．よって，我々の対策技術は，実用的かつ有効な対策技術と言える．

## 7. 課題

Safe Trans ローダは，運用フェーズにおいて新たに脆弱性が見つかった実行バイナリに対して，その攻撃を緩和するパッチの提供を目標とする．現在，本提案手法は Automatic Fortification が置換対象とする関数の一部にしか対応しておらず，より多くの攻撃を防ぐためにも，置換対象関数の拡大が求められる．また，SBoF 攻撃への対策は，フレームポインタを用いたスタック Unwinding を利用し境界検査を行っている．フレームポインタを用いないオプション（`-fomit-frame-pointer`）でコンパイルされた実行バイナリにおいて，提案方式は SBoF 攻撃を緩和できない可能性がある．そのような状況下でも攻撃を防ぐために，今後は，フレームポインタに依存しない強固な Unwinding[47] への対応が必要となる．

表 3　メモリ破壊攻撃への対策技術

| 対策技術 | 対象とする攻撃 | | | 配布済みの実行バイナリへの適用 | 実行時のオーバーヘッド |
|---|---|---|---|---|---|
| | SBoF | HBoF | Use After Free | | |
| **Safe Trans ローダ** | ✓ | ✓ | ✓ | ✓ | 0.64% |
| Stack Smashing Protector | ✓ | | | | 1% |
| Libsafe | ✓ | | | ✓ | 0% |
| Stack Armor | ✓ | | | | 16% |
| Address Sanitizer | ✓ | ✓ | ✓ | | 73% |
| Bounds Checking | ✓ | ✓ | | | 49% |
| CETS | | | ✓ | | 48% |
| DANGNULL | | | ✓ | | 80% |
| FreeSentry | | | ✓ | | 42% |
| Cling | | | ✓ | ✓ | -4% |
| DieHarder | | | ✓ | ✓ | 20% |
| Taint Tracking | ✓ | ✓ | | | 140% |

## 8. まとめ

本論文では，メモリ破壊攻撃を緩和する Safe Trans ローダを提案し，実装と評価を行った．Safe Trans ローダは，メモリ破壊を招くライブラリ関数を，より安全な代替関数に置換することで，これら脆弱性を悪用する攻撃を緩和することができる．本論文の Safe Trans ローダは，メモリ破壊攻撃のうち，SBoF，HBoF，Use After Free 攻撃への対策を行った．また，SPEC CPU2006 ベンチマークを用いて，Safe Trans ローダのパフォーマンスを評価した結果，適用時のオーバーヘッドは 0.64%であることがわかった．

### 参考文献

[1] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, https://cwe.mitre.org/data/definitions/119.html
[2] CWE-121: Stack-based Buffer Overflow, http://cwe.mitre.org/data/definitions/121.html
[3] CWE-122: Heap-based Buffer Overflow, http://cwe.mitre.org/data/definitions/122.html
[4] CWE-416: Use After Free, http://cwe.mitre.org/data/definitions/416.html
[5] PaX Team: Pax address space layout randomization (ASLR) (2003). http://pax.grsecurity.net/docs/aslr.txt
[6] Microsoft: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2 (2008). http://support.microsoft.com/kb/875352
[7] C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proc. of USENIX Security (1998).
[8] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazires, and D. Boneh.Hacking blind. In Proc. of IEEE Security and Privacy (2014).
[9] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In Proc. of IEEE Security and Privacy (2013).
[10] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In Proc. of ACM CCS (2012).
[11] J. Hiser, A. Nguyen-tuong, M. Co, M. Hall, and J. W. Davidson, ILR : Where' d my gadgets go. in Proc. of IEEE Security and Privacy (2012).
[12] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, Shuffler: Fast and deployable continuous code re-randomization. In Proc. of Usenix OSDI (2016).
[13] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlings-son, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In Proc. of USENIX Security (2014).
[14] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. Mc-Camant, D. Song, and W. Zou. Practical control flow in-tegrity & randomization for binary executables In Proc. of IEEE Security and Privacy (2013).
[15] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In Proc. of USENIX Security (2013).
[16] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In Proc. of the 3rd International Workshop on Automatic Debug- ging (1997).
[17] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," In Proc. of ICSE (2006).
[18] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," In Proc. of USENIX Security (2009).
[19] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," In Proc. of SIGPLAN Not (2009).
[20] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," In Proc. of ASIACCS (2010).
[21] CVE-2017-14492, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14492
[22] CVE-2017-14493, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14493
[23] 菅原 捷汰，渡辺 亮平，近藤 秀太，横山 雅展，中村 慈

愛，須崎 有康，齋藤 孝道，主要な Linux ディストリビューションおよびバージョンごとのメモリ破損攻撃への対策技術の適用状況の調査と考察，コンピュータセキュリティシンポジウム (2017).

[24] IPA ISEC セキュア・プログラミング講座：C/C++言語編 第 10 章 著名な脆弱性対策， https://www.ipa.go.jp/security/awareness/vendor/programmingv2/contents/c905.html

[25] John Viega，Gary McGraw，Building Secure Software How to Avoid Security Problems the Right Way, Addison-Wesley 2005

[26] Robert C. Seacord，歌代和正，久保正樹，椎木孝斉，C/C++セキュアコーディング 第 2 版，アスキー・メディアワークス，(2014).

[27] D.Ulrich. Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong). In Proc. of Redhat, April (2009).

[28] ubuntu wiki CompilerFlags, https://wiki.ubuntu.com/ToolChain/CompilerFlags

[29] CVE-2009-2957 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2957

[30] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In Proc. of IEEE Security and Privacy (2013).

[31] A. Baratloo, N. Singh, and T. Tsai. Transparent runtime defense against stack smashing attacks. In Proc. of USENIX Annual Technical Conference (2000).

[32] Linux Programmer's Manual LD.SO(8), http:/man7.org/linux/man-pages/man8/ld.so.8.htm

[33] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In Proc. of NDSS (2015).

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, CETS: compiler enforced temporal safety for CIn Proc. of ISMM (2010).

[35] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing Use-after-free with Dangling Dointers Nullification. In Proc. of NDSS. (2015).

[36] Y. Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In Proc. of NDSS.(2015).

[37] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In Proc. of USENIX Security (2010).

[38] Novark, Gene, and Emery D. Berger. DieHarder: securing the heap. In Proc. of ACMCCS (2010).

[39] E. Bosman, A. Slowinska, and H. Bos, Minemu: the world's fastest taint tracker. In Proc. of RAID (2011).

[40] L. Davi, A.-R. Sadeghi, and M. Winandy, ROPdefender: a detection tool to defend against return-oriented programming attacks. In Proc. of ASIACCS (2011).

[41] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In Proc. of USENIX OSDI (2014).

[42] K. Serebryany, D. Bruening, A. Potapenko, D.Vyukov, Address Sanitizer: A Fast Address Sanity Checker. In Proc. of USENIX conference on Annual Technical Conference (2012).

[43] T. Yamauchi and Y. Ikegami, "HeapRevolver: Delaying and Randomizing Timing of Release of Freed Memory Area to Prevent Use-After-Free Attacks". In Proc. of NSS (2016).

[44] CVE-2013-4256, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4256

[45] SafeStr, https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/safestr

[46] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken.A first step towards automated detection of buffer overrun vulnerabilities. In Proc. of Network and Distributed System Security Symposium (2000).

[47] Dwarf debugging information format, version 4. http://www.dwarfstd.org/doc/ DWARF4.pdf

# 付　　録

## A.1 Safe Trans ローダの置換対象関数について

表 A·1　置換対象の関数

| No | 関数プロトタイプ | 関連する脆弱性 |
|---|---|---|
| 1 | char *strcpy(char *dest, const char *src) | SBoF, HBoF |
| 2 | char *strncpy(char *dest, const char *src, size_t n) | SBoF, HBoF |
| 3 | char *stpcpy(char *dest, const char *src) | SBoF, HBoF |
| 4 | *strcat(char *dest, const char *src) | SBoF, HBoF |
| 5 | char *strncat(char *dest, const char *src, size_t n) | SBoF, HBoF |
| 6 | void *memcpy(void *dest, const void *src, size_t n) | SBoF, HBoF |
| 7 | char *gets(char *s) | SBoF, HBoF |
| 8 | char *getwd(char *buf) | SBoF, HBoF |
| 9 | char *realpath(const char *path, char *resolved_path) | SBoF, HBoF |
| 10 | int sprintf(char *s, const char *format, ...) | SBoF, HBoF |
| 11 | int snprintf(char *s, size_t size, const char *format, ...) | SBoF, HBoF |
| 12 | int vsprintf(char *s, const char *format, va_list ap) | SBoF, HBoF |
| 13 | int vsnprintf(char *s, size_t size, const char *format, va_list ap) | SBoF, HBoF |
| 14 | int scanf(const char *format, ...) | SBoF, HBoF |
| 15 | int fscanf(FILE *stream, const char +format, ...) | SBoF |
| 16 | int sscanf(const char *str, const char *format, ...) | SBoF |
| 17 | int vscanf(const char *format va_list ap) | SBoF |
| 18 | int vsscanf(const char *str, const char *format, va_list ap) | SBoF |
| 19 | int vfscanf(FILE *stream, const char *format, va_list ap) | SBoF |
| 20 | ssize_t read(int fd, void *buf, size_t count) | SBoF |
| 21 | char *fgets(char *s, int len, FILE *stream) | SBoF |
| 22 | void free（void *ptr） | HBoF, Use After Free, Double Free |
| 23 | void *malloc(size_t size) | HBoF |
| 24 | void *calloc(size_t n, size_t size) | HBoF |
| 25 | void *realloc(void *ptr, size_t size) | HBoF |