社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－DPS－126 (40)
2006－CSEC－ 32 (40)
2006／3／17

メモリ保護機構を用いたバッファオーバーフロー検知手法の提案

塩川 真己人† 中里 秀則† 富永 英義†

ソフトウェアのバグは不正アクセスの要因となることがあるが，その中でもバッファオーバーフローの報告件数が圧倒的に多い．バッファオーバーフローを利用した攻撃は，配列に続く何らかのデータを破壊することが目的であり，実際にバッファオーバーフローが起きた瞬間に検知できれば，データの破壊を防ぐことができる．そこで，本研究ではバッファオーバーフローを利用した攻撃を防ぐため，メモリ保護機構を用いてバッファオーバーフローを検知する手法を提案する．

A Buffer Overflow Detection Technique Using Memory Protection Facility

MAKITO SHIOKAWA,† HIDENORI NAKAZATO† and HIDEYOSHI TOMINAGA†

Software bugs can cause intrusion. Overwhelming number of incidents exploiting bugs which cause buffer overflow are reported. The attack exploiting buffer overflow corrupts data which is placed after an array. If it is possible to detect buffer overflow when it occurs, data corruption could be prevented. In this research, we propose a buffer overflow detection technique using memory protection facility to prevent the attack exploiting buffer overflow.

1. はじめに

ソフトウェアのバグは不正アクセスの要因となることがあるが，その中でもバッファオーバーフローの報告件数が圧倒的に多い．バッファオーバーフローとは，確保された配列領域を越えてデータが書き込まれることである．バッファオーバーフローが起きると配列に続くデータが破壊されるため，プログラムは異常動作する．また，攻撃者がバッファオーバーフローを利用し，プログラムの制御に関わるデータを書き換えることで，送り込んだ任意のコードを実行できる場合がある．本研究では，バッファオーバーフローを利用した攻撃を防ぐため，メモリ保護機構を用いてバッファオーバーフローを検知する手法を提案する．

本稿の構成は次の通りである．まず，2 章で提案手法に関連する既存のバッファオーバーフロー対策手法について述べ，3 章で提案手法について説明する．次に，4 章で提案手法の実装方法について述べ，5 章で提案手法の有効性の評価を，6 章で提案手法のオーバーヘッドの評価を示す．さらに，7 章で提案手法のオーバーヘッドを削減する手法について述べる．最後に，8 章でまとめとする．

2. 既存のバッファオーバーフロー対策手法

2.1 propolice

StackGuard[1] は，スタック領域に保存されるリターンアドレスの直前（アドレスの低い側）にカナリアと呼ばれる値を挿入し，バッファオーバーフローによるリターンアドレスの書き換えをカナリアをチェックすることで検知する．propolice[2] は，StackGuard のカナリアの挿入位置をフレームポインタの直前に変更，さらにローカル変数の配置の変更と引数のコピーを行うことで，リターンアドレスに加えフレームポインタ，ローカル変数，引数を保護する．propolice では，カナリアに直接面した配列のオーバーフローを検知することができるが，その他の配列のカナリアに達しないようなオーバーフローを検知することはできない．

2.2 境界チェックコンパイラ

境界チェックコンパイラ[3] は，オブジェクトの境界情報を保持するテーブルを生成し，ポインタ参照と演算の際にポインタが元々指しているオブジェクトの境界を越えていないかチェックを行うことで，バッファオーバーフローを検知することができる．しかし，ポインタ参照と演算の度に境界チェックを行うため，オーバーヘッドが大きい．

2.3 ElectricFence

ElectricFence[4] は，malloc() デバッガであり，ヒープ領域に確保されるメモリブロックにアクセス保護された仮想ページを割り当てることで，バッファオーバーフローを検知する．しかし，メモリの使用量が増加するため，オーバーヘッドが大きい．

3. 提案手法

バッファオーバーフローを利用した攻撃は，配列に続く何らかのデータを破壊することが目的であり，実際にバッファオーバーフローが起きた瞬間に検知できれば，その時点でプログラムを終了するなど適切な処置を行うことで，データの破壊を防ぐことができる．境界チェックコンパイラはこれを実現しているが，全て

† 早稲田大学大学院 国際情報通信研究科
Global Info. and Tele. Studies, WASEDA Univ.

```
void func(char *str)
{
    char buf[128];

    ...
    strcpy(buf, str);
    ...
}
```

図 1 関数 func()

のオブジェクトについて境界チェックを行うため，オーバーヘッドが大きい．さて，propolice では外部からのデータでバッファオーバーフローが発生するのは文字列処理であること，ソースコードでは型ルールに従って文字列が文字型配列に格納されるという仮定のもとに，文字型配列を持たない関数には防御のための変更処理を行わないようにすることでオーバーヘッドを軽減している．本研究においてもこの仮定を受け入れる．従って，バッファオーバーフローを利用した攻撃を防ぐことが目的であれば，文字型配列のバッファオーバーフローを全て検知すれば十分ということになる．そこで，本研究ではスタック領域におけるバッファオーバーフローを利用した攻撃を防ぐため，文字型配列ごとにメモリ保護機構によってアクセス保護された検知領域を割り当てることで，バッファオーバーフローを検知する手法を提案する．

ElectricFence では，malloc() で要求されるメモリブロック全てに保護領域を割り当てるためオーバーヘッドが大きいが，提案手法では，文字型配列のみに選択的に検知領域を割り当てるため，プログラム全体としてのオーバーヘッドを低く抑えられる可能性がある．

3.1 検知領域の割り当て

汎用的な仮想メモリシステムは，ハードウェア (MMU: Memory Management Unit) に基づいてメモリ保護機構を実装している．プログラムの命令が格納されるテキスト領域への書き込みを防止するためなど，プロセスのアドレス空間の一部を保護するため，例えば IA-32 ではページ単位 (4Kbyte) での読み取り/書き込み保護をサポートしている[5]．そこで，このページ単位のアクセス保護を利用し，配列の直後に読み取り専用に設定したページをバッファオーバーフロー検知領域として割り当てる．バッファオーバーフローによって検知領域に書き込みが行われた場合，MMU の保護例外が発生し，プロセスにはセグメンテーション違反のシグナルが送信されるため，バッファオーバーフローが起きた瞬間に検知できるようになる．

3.2 関数に対する処理

3.1 節に述べたように，アクセス保護はページ単位で行われるため，検知領域はプロセスのアドレス空間のページ境界に合わせて割り当てる必要がある．そして，配列は終端が検知領域の直前に位置するように割り当てることになる．ここで，スタック領域上のローカル変数はフレームポインタとのオフセットで参照されるが，関数が呼び出された時のスタックフレームの位置に応じてフレームポインタとページ境界のオフセットは変わるため，配列のオフセットが静的に決定しないことになる．そこで，関数が持つ配列に直接検知領域を割り当てるのではなく，関数が呼び出された時に検知領域と関数が持つ配列に対応する代替配列を確保するようにする．そして，関数本体部ではこの代替配列を使用するようにする．

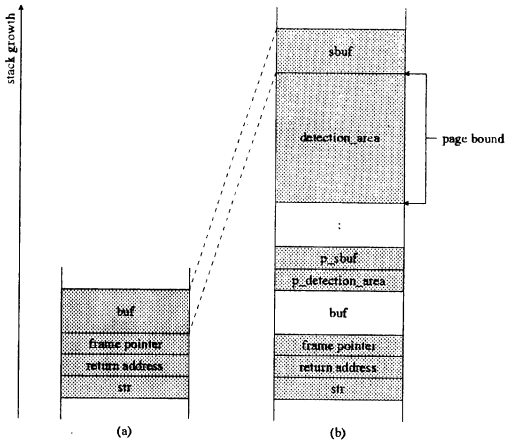

図 3 関数 func() のスタック配置：(a) 適用前，(b) 適用後

説明のために，この処理を図 1 に示す関数 func() に対してソースコードレベルで行ったものを図 2 に示す．また，図 3 に関数 func() の提案手法の適用前と適用後のスタック配置を示す．提案手法では，関数プロローグ部と関数エピローグ部にコードを挿入し，関数本体部のコードを一部変更する．それぞれの部分について次に説明する．

関数プロローグ部

(1) 検知領域のポインタと代替配列のポインタの確保

検知領域のポインタと代替配列のポインタを，関数のローカル変数の直前に確保する．検知領域のアクセス保護をこの検知領域のポインタを通して行う．また，関数本体部における代替配列の参照をこの代替配列のポインタを通して行う．

(2) 検知領域のためのページ境界の計算

現在のスタックトップから最短のページ境界を計算する．

(3) 検知領域の確保

(2) で求めたページ境界からページサイズ (4Kbyte) の検知領域を確保する．

(4) 代替配列の確保

関数が持つ配列に対応する同一サイズの代替配列を確保する．代替配列の終端は検知領域の直前に位置するようになる．

(5) 検知領域を読み取り専用に設定

mprotect() システムコールを使用して検知領域を読み取り専用に設定する．

関数本体部

(6) 関数が持つ配列の参照を代替配列の参照に変更

関数本体部における関数が持つ配列の参照を代替配列の参照に変更する．

関数エピローグ部

(7) 検知領域を読み取り/書き込み可能に設定

```
#define PAGESIZE 4096

unsigned int esp, page_bound;

unsigned int get_esp()
{
    __asm__("movl %esp, %eax");
}

void func(char *str)
{

    /* 関数が持つ配列（提案手法では使用しない） */
    char buf[128];

    /******************************** 関数プロローグ部 ********************************/
    /* (1) 検知領域のポインタ（p_detection_area）と代替配列のポインタ（p_sbuf）の確保 */
    char *p_detection_area, *p_sbuf;
    /* (2) 検知領域のためのページ境界の計算 */
    esp = get_esp();
    page_bound = ((esp + PAGESIZE - 1 ) & ~(PAGESIZE - 1)) - PAGESIZE;
    /* (3) 検知領域の確保 */
    p_detection_area = (char *)alloca(sizeof(char) * ((esp - page_bound) + PAGESIZE));
    /* (4) 代替配列の確保 */
    p_sbuf = (char *)alloca(sizeof(char) * 128);
    /* (5) 検知領域を読み取り専用に設定 */
    mprotect(p_detection_area, PAGESIZE, PROT_NONE);
    /**********************************************************************************/

    /********************************** 関数本体部 **********************************/
    ...
    /* (6) 関数が持つ配列の参照を代替配列の参照に変更 */
    strcpy(p_sbuf, str);
    ...
    /**********************************************************************************/

    /******************************** 関数エピローグ部 ********************************/
    /* (7) 検知領域を読み取り/書き込み可能に設定 */
    mprotect(p_detection_area, PAGESIZE, PROT_READ | PROT_WRITE | PROT_EXEC);
    /**********************************************************************************/
}
```

図 2 提案手法を適用後の関数 func()

mprotect() システムコールを使用して検知領域を読み取り/書き込み可能に設定する．

以上のような処理により，関数において検知領域が割り当てられた配列が使用できるようになる．

3.3 オーバーヘッド

提案手法では，関数プロローグ部と関数エピローグ部に挿入されるコードがオーバーヘッドとなる．そして，関数プロローグ部 (1)〜(5)，関数エピローグ部 (7) の各処理は文字型配列の数だけ繰り返されるため，オーバーヘッドは配列数に比例して増加することになる．また，メモリの使用量の増加もオーバーヘッドとなる．図 4 に関数が複数の文字型配列を持つ場合の提案手法の適用前と適用後のスタック配置を示す．提案手法では，検知領域が 1 ページ，代替配列がサイズに関わらず少なくとも 1 ページを使用するため，ページの割り当てと参照のためのオーバーヘッドが増加する．6 章で，提案手法のオーバーヘッドの評価を示す．

4. 実　　装

提案手法を GCC に実装した．GCC ではソースコードから構文木を生成し，RTL (Register Transfer Language) と呼ばれる中間コードに変換する．そして，実装では 3.2 節に述べたような関数プロローグ部と関数エピローグ部へのコードの挿入と関数本体部のコードの変更を RTL レベルで行うようにした．現在の実装はプロトタイプであり，正常にコンパイル・実行する

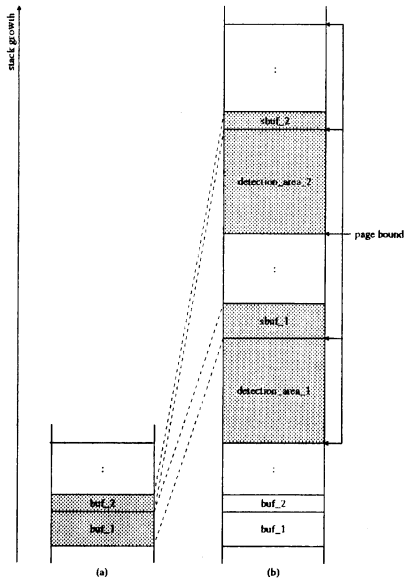

図 4 関数が複数の文字型配列を持つ場合のスタック配置：(a) 適用前，(b) 適用後

```
int main()
{
    char buf_1[8];
    char buf_2[8];

    strcpy(buf_1, "1234567")
    strcpy(buf_2, "xxxxxxxxxxx");

    printf("%s\n", buf_1);

    return 0;
}
```

図 5 サンプルプログラム

表 1 実行結果

プログラム	適用前	適用後
サンプルプログラム	"xxx" の出力	強制終了
SuperProbe	root shell の起動	強制終了

ことができるプログラムは限られている．

5. 有効性の評価

提案手法の有効性を評価するため，バッファオーバーフロー脆弱性を持つプログラムの提案手法の適用前と適用後の実行結果を比較した．バッファオーバーフロー脆弱性を持つプログラムとして，図 5 に示すサンプルプログラムと，XFree86 2.1 に付属する SuperProbe を使用した．このとき，サンプルプログラムについてはサンプルプログラムを実行した時の結果を，SuperProbe については SuperProbe に対して exploit（攻撃コード）を実行した時の結果を比較した．表 1 に実行結果を示す．

まず，サンプルプログラムについて，提案手法の適用前では，buf_2 のオーバーフローによって buf_1 に格納されている文字列の一部が書き換えられ，buf_1 の内容の出力は "xxx" となった．それに対して，提案手法の適用後では，buf_2 のオーバーフローによってプログラムはセグメンテーション違反を起こし，強制終了した．この結果から，提案手法によって文字型配列のオーバーフローを全て検知できることが確認できた．

次に，SuperProbe について，提案手法の適用前では，exploit の実行によって root shell が起動した．それに対して，提案手法の適用後では，exploit の実行によってプログラムはセグメンテーション違反を起こし，強制終了した．この結果から，提案手法によって実際にバッファオーバーフローを利用した攻撃を防げることが確認できた．

```
void test()
{
    char buf_1[BUF_SIZE];
    char buf_2[BUF_SIZE];
    char buf_3[BUF_SIZE];
    ...

    strcpy(buf_1, "1234567");
    strcpy(buf_2, "1234567");
    strcpy(buf_3, "1234567");
    ...
}
```

図 6 関数 test()

表 2 測定環境

OS	Fedora Core 1(Linux Kernel 2.4.22-1)
CPU	AMD(R) Duron(R) 800MHz
メモリー	128Mbyte

6. オーバーヘッドの評価

6.1 オーバーヘッドの最大値

提案手法のオーバーヘッドの最大値を評価するため，図 6 に示す関数 test() の提案手法の適用前と適用後の実行時間を比較した．このとき，表 2 に示す環境で，関数 test() を 100,000 回呼び出した時の CPU 時間を測定し，比較した．提案手法では関数が持つ配列数によってオーバーヘッドが変化するため，関数 test() の配列数が 10 個と 100 個の場合を測定した．また，それぞれの場合で，配列サイズ（BUF_SIZE）が 8，128，512，1024，4096byte の場合を測定した．表 3 に配列数が 10 個の場合，表 4 に 100 個の場合の測定結果を示す．ここで，表中のオーバーヘッドとは提案手法の適用前と適用後の CPU 時間（user CPU 時間と system CPU 時間の合計）の比である．

配列数が 10 個の場合，配列サイズの違いによるオーバーヘッドの変化はそれほどなく，およそ 250 倍前後となった．配列数が 100 個の場合，配列サイズが大きくなるほどオーバーヘッドは小さくなり，配列サイズが 8byte の場合でおよそ 550 倍，配列サイズが 4096byte

表 3 測定結果：関数 test() (配列数 10 個)

BUF_SIZE(byte)	適用前			適用後			オーバーヘッド
	user(sec)	system(sec)	total(sec)	user(sec)	system(sec)	total(sec)	
8	0.080	0.030	0.110	0.420	26.520	26.940	245
128	0.080	0.020	0.100	0.620	26.290	26.910	269
512	0.080	0.020	0.100	0.500	26.180	26.680	267
1024	0.090	0.020	0.110	0.550	26.160	26.710	243
4096	0.090	0.010	0.100	0.520	26.250	26.770	268

表 4 測定結果：関数 test() (配列数 100 個)

BUF_SIZE(byte)	適用前			適用後			オーバーヘッド
	user(sec)	system(sec)	total(sec)	user(sec)	system(sec)	total(sec)	
8	0.540	0.010	0.550	5.310	296.160	301.470	548
128	0.550	0.020	0.570	6.610	295.510	302.120	530
512	0.540	0.020	0.560	5.470	294.690	300.160	536
1024	0.580	0.050	0.630	6.220	291.230	297.450	472
4096	3.140	0.040	3.180	5.010	285.260	290.270	91

表 5 測定結果：SuperProbe

適用前			適用後			オーバーヘッド
user(sec)	system(sec)	total(sec)	user(sec)	system(sec)	total(sec)	
0.087	0.021	0.108	0.089	0.023	0.112	1.04

の場合でおよそ 90 倍となった．関数 test() では，関数本体部のコードの実行コストに比べ，提案手法を適用することによって関数プロローグ部と関数エピローグ部に挿入されるコードの実行コストが明らかに大きなものになるため，これらのオーバーヘッドの値はそれぞれの配列数における最大値といえる．

6.2 プログラム全体としてのオーバーヘッド

提案手法では，関数が文字型配列を持たない場合，提案手法によるオーバーヘッドを無視できる．従って，複数の関数が定義されているような実プログラムでは，プログラム全体としてのオーバーヘッドはもっと低く抑えられると考えられる．そこで，5 章で使用した SuperProbe の提案手法の適用前と適用後の実行時間を測定し，比較した．表 5 に測定結果を示す．

オーバーヘッドはおよそ 1.04 倍となった．SuperProbe のソースコードを調べたところ，測定環境においてコンパイルされる関数 107 個のうち，文字型配列を持つものは 2 個であり，いずれの関数でも文字型配列の数は 1 個であった．この結果から，プログラムによっては提案手法によるオーバーヘッドを低く抑えられることが確認できた．

7. オーバーヘッドの削減

提案手法のオーバーヘッドを削減するため，関数プロローグ部 (5) と関数エピローグ部 (7) で使用している mprotect() システムコールを提案手法に最適化する．

7.1 システムコールの最適化

提案手法では，検知領域のアクセス保護を行うのに mprotect() システムコールを使用している．ここで，図 7 に mprotect() システムコールの書式を示す．mprotect() システムコールは，単一区間 [addr,addr+len-1] に prot に指定したアクセス保護を行う．従って，提案手法では関数が n 個の文字型配列を持っている場合，n 個の検知領域のアクセス保護を行うために，図 8 に示すように n 回 mprotect() システムコールを呼び出す必要がある．このとき，プロセスのユーザ・モードとカーネル・モードとの間のモード移行も n 回行われることになる．

さて，提案手法が適用された関数では，ローカル変

```
int mprotect(const void *addr, size_t len, int prot);
```

図 7 mprotect() システムコールの書式

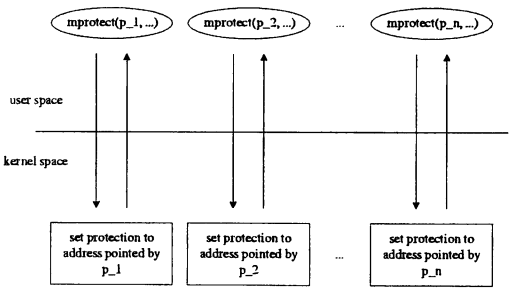

図 8 mprotect() システムコールを使用した n 区間のアクセス保護

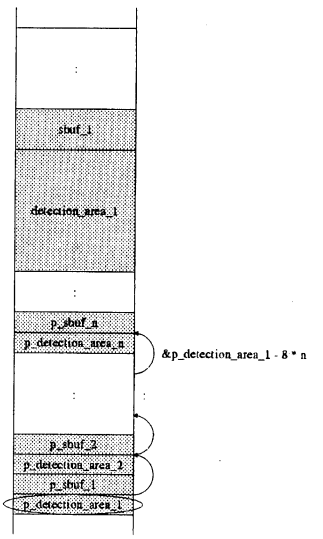

図 9 n 番目の検知領域のポインタのアドレスの取得

表 6 測定結果：関数 test() (配列数 10 個，opt_mprotect() システムコール)

BUF_SIZE(byte)	適用前			適用後			オーバーヘッド
	user(sec)	system(sec)	total(sec)	user(sec)	system(sec)	total(sec)	
8	0.080	0.030	0.110	0.060	23.780	23.840	217
128	0.080	0.020	0.100	0.090	23.400	23.490	235
512	0.080	0.020	0.100	0.060	23.620	23.680	237
1024	0.090	0.020	0.110	0.050	23.320	23.370	215
4096	0.090	0.010	0.100	0.060	23.680	23.740	237

表 7 測定結果：関数 test() (配列数 100 個，opt_mprotect() システムコール)

BUF_SIZE(byte)	適用前			適用後			オーバーヘッド
	user(sec)	system(sec)	total(sec)	user(sec)	system(sec)	total(sec)	
8	0.540	0.010	0.550	1.150	266.170	267.320	486
128	0.550	0.020	0.570	1.120	266.600	267.720	470
512	0.540	0.020	0.560	0.930	255.030	255.960	457
1024	0.580	0.050	0.630	0.820	251.900	252.720	401
4096	3.140	0.040	3.180	1.020	253.080	254.100	80

```
int opt_mprotect(int num, const void **addr, size_t len, int prot);
```

図 10 opt_mprotect() システムコールの書式

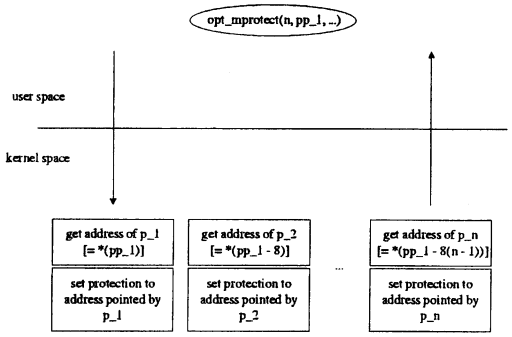

図 11 opt_mprotect() システムコールを使用した n 区間のアクセス保護

数の直前に検知領域のポインタと代替配列のポインタが文字型配列の数だけ規則的に確保される．これを利用すると，図 9 に示すように 1 つ目の検知領域のポインタのアドレスから 2 つ目以降の検知領域のポインタのアドレスを辿ることができる．そこで，システムコール引数として 1 つ目の検知領域のポインタのアドレスを取り，カーネル内で 2 つ目以降の検知領域のアドレスを辿り，全ての検知領域のアクセス保護を行う，opt_mprotect() システムコールを実装した．図 10 に opt_mprotect() システムコールの書式を示す．num はアクセス保護を行う区間数である．opt_mprotect() システムコールでは，n 個の検知領域のアクセス保護を行う場合でも，図 11 に示すようにモード移行が 1 回で済むようになるため，モード移行によるオーバーヘッドを削減することができる．

7.2 オーバーヘッドの再評価

mprotect() システムコールの代わりに opt_mprotect() システムコールを使用することでオーバーヘッドの最大値がどれだけ低下するか 6.1 節と同様にして評価した．表 6 に配列数が 10 個の場合，表 7 に 100 個の場合の測定結果を示す．

配列数が 10 個の場合，オーバーヘッドはおよそ 220 倍前後，配列数が 100 個の場合，およそ 80〜490 倍となった．いずれの配列数の場合でも，オーバーヘッドの最大値は 1 割強低下した．

8. ま と め

本研究では，スタック領域におけるバッファオーバーフローを利用した攻撃を防ぐため，文字型配列ごとにメモリ保護機構によってアクセス保護された検知領域を割り当てることで，バッファオーバーフローを検知する手法を提案した．そして，提案手法の有効性をバッファオーバーフロー脆弱性を持つプログラムを使用して示した．また，提案手法のオーバーヘッドの最大値は配列数に応じて数 10〜数 100 倍となったが，プログラムによってはオーバーヘッドを低く抑えられることを示した．従って，今後の課題としては実装を完全なものにし，幅広いプログラムにおいてオーバーヘッドを評価することが挙げられる．

参 考 文 献

1) Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,"In 7th USENIX Security Conference, pages 63-77, San Antonio, TX, 1998

2) 江藤博明, 依田邦和, "propolice: スタックスマッシング攻撃検出手法の改良," 情報処理学会 研究報告「コンピュータセキュリティ」Vol.2001 No.075

3) Richard Jones, Paul Kelly, "Bounds Checking for C,"http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html

4) "ElectricFence,"http://perens.com/FreeSoftware/ElectricFence/

5) "IA-32 インテル®アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル"