計算機言語システム論 II
(2005/11/14)

米澤研究室 M1 佐藤秀明

本日の題材

- Efficient Techniques for Comprehensive Protection from Memory Error Exploits
  - Sandeep Bhatkar, R. Sekar and Daniel C. DuVarney
- メモリエラーを突く攻撃を高確率で無力化する手法
  - メモリを使用する方法にランダム性を導入

メモリエラーの脅威

- メモリへの不正アクセスが大人気
  - US-CERT アドバイザリの 80% 以上
    - stack smashing
    - heap overflow
    - format string attack
- Stack smashing 以外の新しい攻撃の増加
  - 近いうちに攻撃法の主流となる可能性

メモリエラーを防ぐ既存技術

- StackGuard
  - canary word の導入
  - 防げるのは stack smashing だけ
- CRED
  - 境界チェック
  - 高いオーバヘッド
- CCured 、Cyclone
  - C との互換性なし
- 完璧さと導入しやすさのトレードオフ

目標設定

- 究極的には：全ての不正なメモリアクセスを防ぐ
  - 速度や互換性を犠牲にする覚悟
- とりあえず：攻撃者の意図した挙動をとらない
  - メモリを不正に使われること自体は不問
- 具体策：攻撃された後の挙動を unpredictable に
  - 攻撃の成功率を限りなくゼロに近づける
  - 互換性を保ちながら低コストで導入可能 (のはず)

Address Space Randomization

- ASR: メモリの使い方をランダム化する技術
  - コード / データの位置を毎回変更
- 既存の ASR はすべてのメモリ攻撃に対応できない
  - 相対アドレスを利用する攻撃
  - information leakage attack
  - ランダム化そのものに対する攻撃
- 本論文の目的 : すべてのメモリ攻撃に対応する ASR

アプローチの概要

- メモリ上のデータ / コードの位置をランダム化
  - 絶対位置
  - 2 つのオブジェクト間の相対位置
- 自己ランダム化プログラムの導入
  - ソース変換により組み込む
  - ロード時や実行中に発動

メモリエラーの種類

- Spatial error
  - 本来指すべきオブジェクトの外を指すエラー
- Temporal error
  - 既に free されたオブジェクトの位置を指すエラー
- 各エラーによる挙動を予測不能にできるか？

Spatial Error への対処

- 非ポインタ型データの dereference: 予測不能
  - 同じポインタ値が毎回異なるオブジェクトを指す
- 未初期化ポインタの dereference: 予測不能
  - call ごとに毎回異なる位置に確保
    - ローカル変数
    - ヒープ上のオブジェクト
- 不正なポインタ演算：予測不能
  - 他のオブジェクトとの相対位置は毎回変更

Temporal Error への対処

- Reallocate された位置への dereference: 予測不能
  - オブジェクトの alloc される位置は毎回変更

我々のアプローチの利点

- 容易な導入
  - 自動ソース変換
  - レガシーコードとの互換性
  - 環境非依存
  - 同一コピーの配布
- 包括的なランダム化
  - 相対位置を利用した攻撃にも対応
- 小さい実行時オーバヘッド

静的データの変換 (1/4)

- 実行開始時に各変数の位置を決定
  - 実行時はポインタを通じてアクセス
- 関数 static_alloc() を用意

1. mmap で大きな領域を確保
   - ベースアドレスを自由にとれる
2. 各変数の領域を確保する順序をランダム化
3. バッファ型領域の前後に書き込み不可のランダムギャップを挿入
4. 割り当てられた各領域をゼロクリア
5. 各領域を指すポインタをセット

静的データの変換 (2/4)

- コード変換前

```
int a = 1;
char b[100];
extern int c;

void f() {
    while (a < 100) b[a] = a++;
}
```

静的データの変換 (3/4)

- 変換後

```
int *a_ptr;
char (*b_ptr) [100];
extern int *c_ptr;

void f() {
        while ((*a_ptr) < 100)
          (*b_ptr)[(*a_ptr)] = (*a_ptr)++;
}
```

静的データの変換 (4/4)

- 初期化コード (プログラムの最初に実行)
  - 構造体 alloc_info で変数の情報を渡す

```
void __attribute__ ((constructor)) data_init(){

    alloc_info[0].ptr = (void *) &a_ptr;
    alloc_info[0].size = sizeof(int);
    alloc_info[0].is_buffer = FALSE;
    alloc_info[1].ptr = (void *) &b_ptr;
    alloc_info[1].size = sizeof(char [100]);
    alloc_info[1].is_buffer = TRUE;

    static_alloc(alloc_info, 2);

    (*a_ptr) = 1;
}
```

コードの変換 (1/3)

- 関数単位で配置をシャッフル
  - 実行時は関数ポインタを通じて呼び出し
  - 各関数間は書き込み禁止のランダムギャップで埋める
- ソースレベルでは関数ポインタのテーブルを用意
  - バイナリレベルで関数配置をランダム化、テーブル更新
    - プログラム実行時のランダム化は実装していない
    - ソース変換だけで済むんじゃなかったのか？
  - テーブルの前後を特殊なマークで囲む
    - テーブルの位置を簡単に見つけるため
- 絶対アドレスを用い得る要素はソースレベルで消去
  - switch 文は if-then-else に変換

コードの変換 (2/3)

- 変換前ソース

```
char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = &f;
  str = (*fptr)();
  g(10);
}
```

コードの変換 (3/3)

- 変換後ソース

```
void *const func_ptrs[] =
  {M1, M2, M3, M4, (void *)&f, (void *)&g,
   M5, M6, M7, M8};

char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = (char *(*)())func_ptrs[4];
  str = (*fptr)();
  (*((void (*)(int)) (func_ptrs[5])))(10);
}
```

- コンパイル後にバイナリ変換

スタックの変換

- バッファ型のローカル変数は shadow stack に確保
  - リターンアドレスやポインタ型変数の書き換えを防止
  - 関数 shadow_alloc() を用意
    - call ごとにバッファの確保順序を変更
- スタックのベースアドレスをランダム化
  - スタックの底 (バイナリ変換)
  - フレーム間のギャップ (関数 alloca() を呼び出し)

ヒープの変換

- 各オブジェクト間に書込み禁止のランダムギャップ
  - brk() システムコール
- 各オブジェクトのサイズを 30% までランダムに拡大
  - malloc() へのラッパー関数を挟む

DLL の変換

- 再配置可能オブジェクトとジャンクコードをランダムに混ぜて DLL を作成
  - システムをリブートするたびに作り直すとよい
- ダイナミックリンカがランダムな位置にロード

その他の変換

- 動的リンク情報
  - 攻撃者に狙われやすい情報
    - ジャンプ先を書き換えられる
- Global Offset Table(外部アドレスのテーブル)
  - 開始時に全てリンクしてすぐ書き込みを禁止
- Procedure Linkage Table(外部関数の呼び出しルーチン)
  - 通常の関数と同様に配置をランダム化

性能評価 (1/4)

- 平均 11% の実行時オーバヘッド

Program	Orig. CPU time	% Overheads			
		Stack	Static	Code	All
grep	0.33	0	0	0	**2**
tar	1.06	2	2	1	**4**
patch	0.39	2	0	0	**4**
wu-ftpd	0.98	2	0	6	**9**
bc	5.33	7	1	2	**9**
enscript	1.44	8	3	0	**10**
bison	0.65	4	0	7	**12**
gzip	2.32	6	9	4	**17**
sshd	3.77	6	10	2	**19**
ctags	9.46	10	3	8	**23**
Avg. Overhead		**5**	**3**	**3**	**11**

性能評価 (2/4)

- 静的変数へのアクセスが多いほど遅くなる
  - ポインタを通したアクセス

Program	%age of variable accesses		
	Local		Global
	(non-buffer)	(buffer)	(static)
grep	99.9	0.004	0.1
bc	99.3	0.047	0.6
tar	96.5	0.247	3.2
patch	91.8	1.958	6.2
enscript	90.5	0.954	8.5
bison	88.2	0.400	10.9
ctags	72.9	0.186	26.9
gzip	59.2	0.018	40.7

性能評価 (3/4)

- Call 回数が多いほど遅くなる
  - 関数ポインタ、shadow stack 、ギャップ挿入

Program	# calls	calls/ sec.	Shadow stack allocations	
	$\times 10^6$	$\times 10^6$	per sec.	per call
grep	0.02	0.06	24K	0.412
tar	0.43	0.41	57K	0.140
bison	2.69	4.11	423K	0.103
bc	22.56	4.24	339K	0.080
enscript	9.62	6.68	468K	0.070
patch	3.79	9.75	166K	0.017
gzip	26.72	11.52	0K	0.000
ctags	251.63	26.60	160K	0.006

性能評価 (4/4)

- Apache はほぼオーバヘッドなし

#clients	Degradation (%)	
	Connection Rate	Response Time
2-clients	1	0
16-clients	0	0
30-clients	0	1

メモリエラー攻撃に対する効果

- 攻撃成功率 =P(Owr)*P(Eff)
  - P(Owr)= 攻撃者が希望のデータを上書きできる確率
  - P(Eff)= 上書きされたデータが攻撃者の意図する効果をもたらす確率
- ただし以下のいずれかを仮定
  - 攻撃失敗後は再びランダム化操作が行われる
  - 失敗したのが上書きかその後の効果発揮かは攻撃者には分からない
- 両方とも不成立の場合
  - 攻撃成功率 =min(P(Owr),P(Eff))

P(Owr) の評価 (1/2)

- Stack buffer overflow
  - バッファ型変数は shadow stack に隔離してるから無理
  - となりのバッファを上書きする攻撃は可能だが困難
    - バッファ内に critical なデータは少ない
    - そもそもスタックにとられるバッファが少ない
- Static buffer overflow
  - 非常に困難
    - 各オブジェクト間の位置関係は予測不能
    - 各オブジェクト間には書き込み禁止エリアが存在

P(Owr) の評価 (2/2)

- Heap overflow
  - ヒープブロックを繋ぐポインタは依然上書きされやすい
    - 双方向連結リスト
  - しかし P(Eff) は非常に低い
    - GOT の絶対位置は予測不能
- Format string attack
  - 書き換え先のアドレスを自由に指定できない
    - string は shadow stack にあるから

P(Eff) の評価

- 非ポインタ型データの書き換え
  - うまく書き換えできたらその時点で目的達成
    - ユーザ ID など
  - ただし我々の手法は P(Owr) が小さいので大丈夫
- ポインタの書き換え
  - 希望の要素をうまく指し当てるのは困難
    - 現存データへのポイント
    - 攻撃者が注入したデータへのポイント
    - 現存コードへのポイント
    - 攻撃者が注入したデータへのポイント
  - 相対位置すら予測不能だとほとんど無理

ランダム化を考慮した攻撃

- Information leakage attack
  - 何らかの方法でプログラムの配置情報を入手
    - 特定の要素へのポインタ
  - 入手した情報を利用してさらに overflow を起こす必要性
    - 我々の手法に対しては無力だろう
- Brute force and guessing attack
  - 常時公開されているシステムの場合
  - 我々の手法を打ち負かす攻撃法はまだないからいいや
- Partial pointer overwrite
  - ポインタの下位ビットのみ書き換えて境界チェックを突破
  - プログラムの配置をランダム化してるから大丈夫

関連研究 (1/2)

- リターンアドレスの保護
  - canary word の導入
    - Stackguard
  - リターンアドレスのバックアップをとる
    - Stackshield 、RAD
  - ライブラリ形式で再コンパイルなしに導入可能
    - Libsafe 、Libverify
- ポインタの保護
  - スタックレイアウトの改良
    - ProPolice
  - ポインタの「暗号化」
    - PointGuard

関連研究 (2/2)

- Format string attack への対応
  - ソース変換
    - FormatGuard
- 不正なメモリアクセスのチェック
  - 実行時…高い実行時間オーバヘッド
  - コンパイル時…高い false positive
- プログラム実行のランダム化
  - Address randomization
  - Operating system functions randomization
  - Instruction set randomization

まとめ

- Address Space Randomization の改良
  - 相対アドレスを利用した攻撃の防止
- ランダム化の粒度を縮小
  - 関数単位、静的変数単位
  - バッファ型変数の隔離
- 高い利便性
  - 小さい実行時間オーバヘッド
  - 広範な種類の攻撃に対する耐性