Buffer Overflow & SQL Injection

Università degli Studi di Udine
Dipartimento di Ingegneria Gestionale,
Elettrica e Meccanica

21 Marzo 2011

Scaletta

Memory layout in x86

Quando, in una architettura x86 (e.g. AMD Athlon, Intel Core 2, Pentium 4, i7, ...), viene eseguito un programma, il sistema operativo riserva una certa quantità di memoria RAM per la sua esecuzione.

La memoria di un programma in esecuzione è suddivisa in

- **data segment**, spazio dedicato a variabili *globali*, *statiche* e *dinamiche non locali* (heap),
- **text segment**, spazio per il codice macchina da eseguire, e
- **stack**, spazio per variabili dinamiche locali.

Memory layout in x86

Mentre la **heap**, che viene gestita tramite le chiamate a `malloc` e `free`, cresce dagli indirizzi di memoria più bassi a quelli più alti, lo **stack** cresce in direzione inversa.

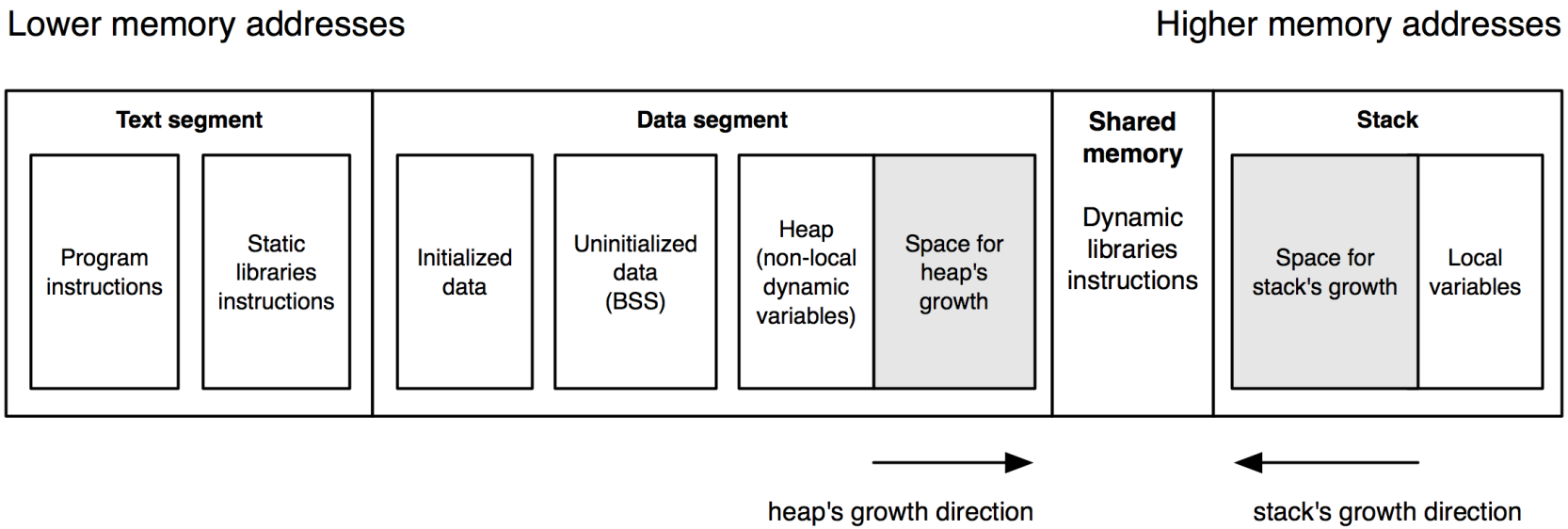


Figure: Process memory layout in x86

Stack layout in x86

Lo **stack** è la chiave per la gestione delle variabili locali alle funzioni nei linguaggi di programmazione di alto livello (C, C++, ...). Esso è una LIFO (*last-in-first-out*), ovvero una collezione in cui l'ultimo elemento aggiunto (*push*) è anche il primo ad essere rimosso (*pull*).

Quando in C viene chiamata una funzione, sullo stack viene allocato un nuovo **stack frame**.

Stack layout in x86, cont'd

Uno stack frame contiene:

- i parametri passati alla funzione,
- le variabili locali dichiarate nel corpo della funzione,
- l'indirizzo della prossima istruzione (text segment).

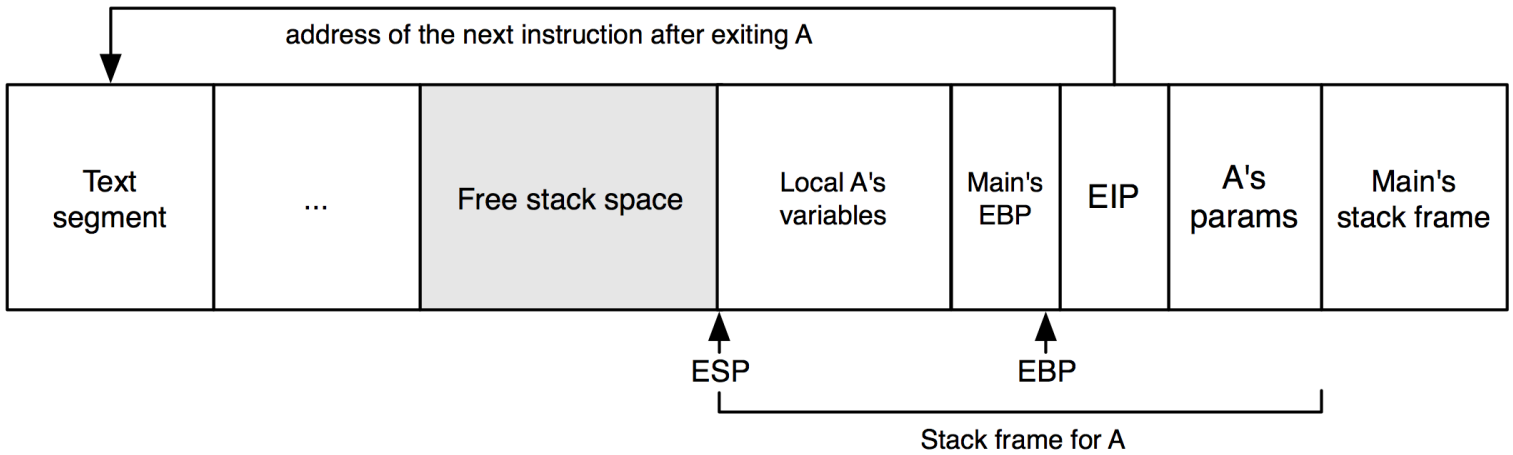


Figure: Stack frame layout in x86, simplified

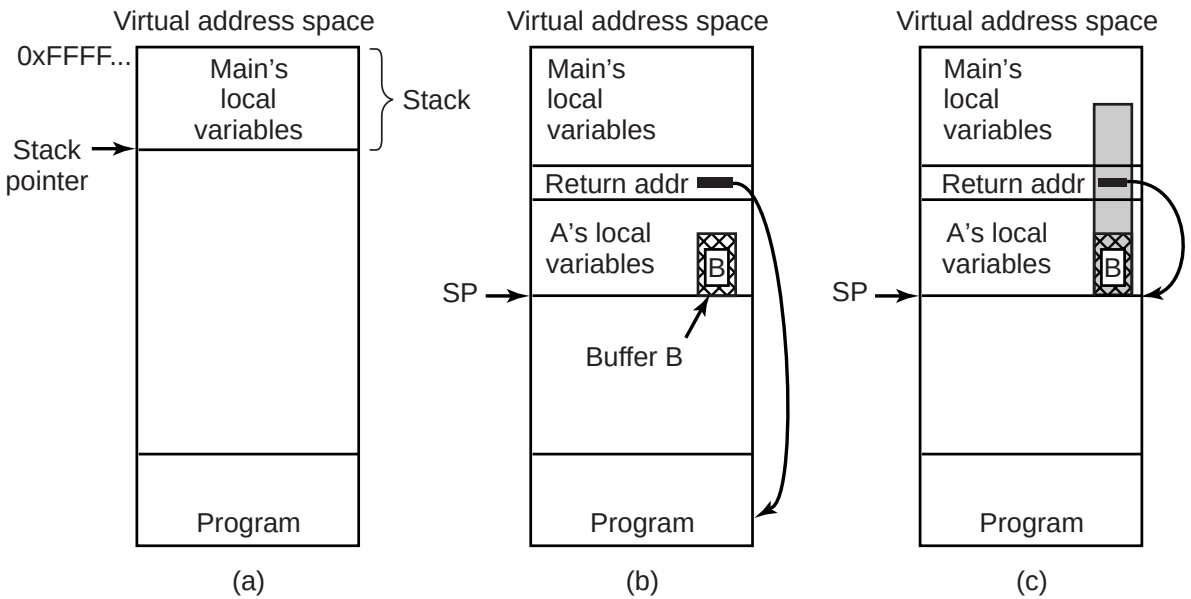
Esempio

Esempio di funzionamento di una chiamata di funzione:

```
1  void A() {
2      char B[50];
3  }
4
5  void main() {
6      A();
7      printf("Done.\n");
8  }
```

Esempio, cont'd

Prima della chiamata, l'ultimo stack frame è quello di `main` (a). Al momento della chiamata di `A` (b), il suo stack frame viene allocato ed EIP (*Return addr* in figura) punta al codice macchina per la chiamata a *printf*.



Buffer e limiti di capacità

Con il termine **buffer** si indica una regione di memoria contigua (i cui indirizzi sono consecutivi) che può essere utilizzata per memorizzare temporaneamente delle variabili. Un esempio di buffer sono gli array in C.

In C e C++ (i linguaggi più usati nella realizzazione di sistemi operativi) non viene effettuato alcun controllo sulle dimensioni dei buffer. Per esempio, è possibile scrivere in un array più elementi di quelli per il quale è stato dichiarato.

Buffer overflow

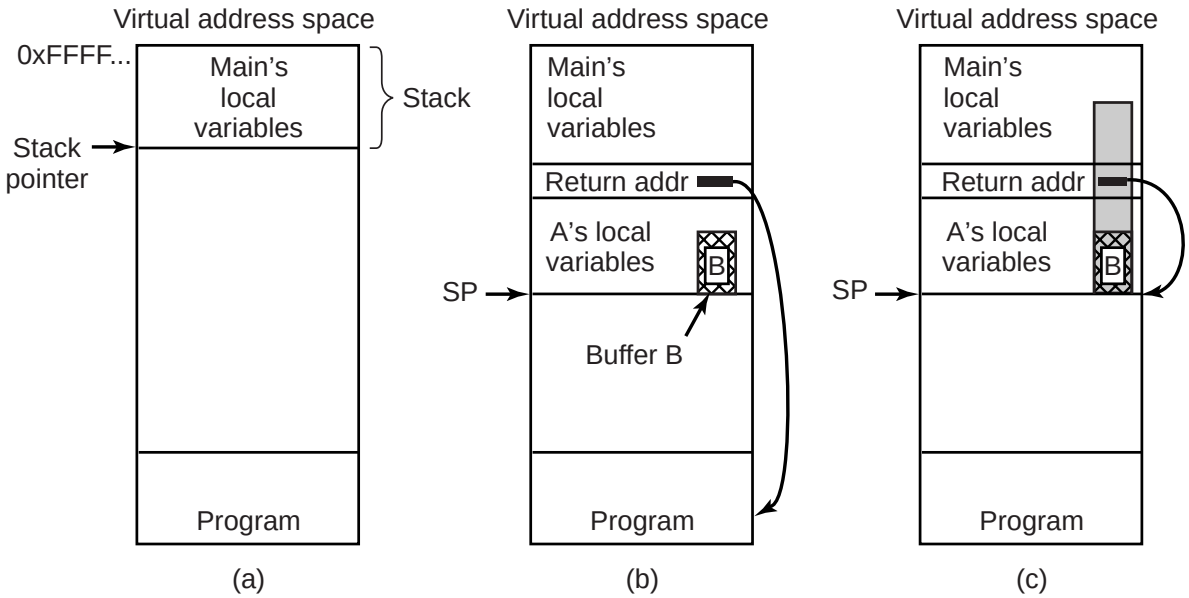
Esempio:

```
1 void main() {
2       char B[5];
3       B[5] = 'a';
4 }
```

`B` può contenere 5 elementi, quindi gli indici vanno da 0 a 4 perché in C si inizia a contare da zero. Dove verrà scritta la 'a'? Riguardiamo lo schema della chiamata di funzione.

Buffer overflow, cont'd

Ci troviamo nel caso (c), il contenuto dell'array sfora i limiti di capacità e, poichè lo stack cresce dagli indirizzi più alti a quelli più bassi, ogni valore di troppo sovrascriverà la memoria precedente a `B`. Questa situazione è detta buffer overflow.



Buffer overflow, cont'd

Se prima di `B` sono state dichiarate delle **altre variabili**, queste verranno sovrascritte.

Poi verrà sovrascritto il **puntatore allo stack frame** precedente.

Infine verrà sovrascritto l'**indirizzo di ritorno**, ovvero l'indirizzo della prossima istruzione da eseguire.

Attacco buffer overflow

L'**attacco buffer overflow** consiste nel sovrascrivere l'indirizzo di ritorno con l'indirizzo di una porzione di memoria contenente del codice macchina eseguibile.

Questo attacco è stato documentato per la prima volta nel 1988, ed è stato descritto approfonditamente in un famoso articolo del 1996 intitolato *Smashing the stack for fun and profit* (vedere riferimenti).

L'obiettivo (generalmente) è ottenere una shell con privilegi da amministratore.

Esempio

Prima di tutto bisogna individuare un software vulnerabile. Particolarmente sono noti i software che utilizzano la funzione:

```
char *strcpy(char *destination, const char *source)
```

Poichè essa non effettua alcun controllo sulla lunghezza di `source` e `destination`, essa è prona a buffer overflow. Un esempio di software come questo è il semplice programma `vulnerable.c` (che trovate nella directory Code/BufferOverflow sulla vostra macchina) che stampa in output l'input fornito.

Esempio, cont'd

Una volta compilato

```
gcc -z execstack -fno-stack-protector -o vulnerable vulnerable.c
```

il programma può essere eseguito con il comando:

```
$ ./vulnerable testo_in_input
```

Se `testo_in_input` rientra nella dimensione del buffer, otterrete in output il testo stesso. Altrimenti il programma incontrerà un buffer overflow e terminerà con un errore **Segmentation fault**.

Esempio, cont'd

Il programma `exploit.c` sfrutta la vulnerabilità di `vulnerable` per eseguire del codice malevolo in grado di restituire una shell da amministratore. Il funzionamento di `exploit` è il seguente:

1. genera un ambiente di esecuzione per il programma `vulnerable`, contenente, sotto forma di stringa, il codice macchina per ottenere lo `uid` 0 (l'indentità dell'amministatore) e aprire una shell,
2. genera un parametro per `vulnerable` più lungo di quello ammesso, la cui porzione finale è riempita con l'indirizzo di ritorno del codice macchina eseguibile.
3. invoca `vulnerable` passando l'ambiente e il parametro generato.

Esempio, cont'd

Ci sono alcuni limiti:

1. per eseguire `setuid(0)`, il programma vulnerabile deve essere di proprietà dell'amministratore (root),
2. i moderni compilatori utilizzano dei meccanismi di protezione contro gli attacchi di questo tipo, quindi se il programma vulnerabile è compilato senza particolari opzioni, non sarà vulnerabile (per questo usiamo `-z execstack -fno-stack-protector`),
3. i moderni sistemi operativi utilizzano dei meccanismi di protezione della memoria, che devono essere disabilitati per poter far funzionare questo esempio.

Esempio, cont'd

Per semplificare questa procedura useremo un Makefile che automatizzerà la procedura di compilazione disabilitando le protezioni e rendendo `vulnerable` di proprietà dell'utente root. Per compilare l'esempio sarà quindi sufficiente digitare:

```
make
```

Per effettuare un vero attacco di questo tipo è necessario individuare un programma sul sistema che sia già di proprietà di root e che soffra della vulnerabilità a buffer overflow (e.g. utilizza `strcpy`).

Esempio, cont'd

A questo punto, per eseguire l'attacco è sufficiente mandare in esecuzione `exploit`:

```
$ whoami
guy
$ ./exploit
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ...
# whoami root
#
```

I caratteri 'A' sono quelli con cui abbiamo riempito il parametro fino alla lunghezza del buffer, i caratteri seguenti (non stampabili) sono i codici delle operazioni necessarie (a basso livello) ad aprire la shell.

Shellcode

Non abbiamo analizzato come sia possibile costruire il codice macchina per assumere l'identità dell'amministratore e aprire la shell. La procedura è un po' macchinosa complicata e consiste nelle seguenti fasi:

- Si scrive un programma C che esegua `setuid(0)` e `execve("/bin/sh")`,
- lo si compila sull'architettura target con linking statico,
- si decompila il codice e si legge il codice assembly,
- si converte l'assembly in codice macchina eliminando eventuali caratteri nulli, che spezzerebbero la stringa C che lo contiene.

Ci sono dei software che automatizzano questa procedura (un esempio è `generator.c` nella directory Code/ShellCodeGenerator).

Contromisure

Abbiamo accennato prima al fatto che esistono misure di sicurezza per evitare attacchi di questo tipo. Alcune di esse sono:

- **Executable space protection**, il contenuto dello stack viene marcato in in modo tale che non possa essere eseguito, questi metodi possono essere implementati nel sistema operativo (e.g. PaX in Linux) o sul processore stesso (e.g. bit NX),
- **Address space layout randomization (ASLR)** gli indirizzi della memoria di un processo vengono assegnati in modo casuale, impedendo quindi di fare assunzioni sulle posizioni delle variabili, rende l'attacco improbabile ma non impossibile,

Contromisure, cont'd

- **Canarini** (StackGuard, ProPolice) implementati nei compilatori (GCC, Visual Studio, ...), sono dei valori di guardia che vengono messi attorno ai buffer, il principio è che se il valore del canarino cambia c'è in atto un buffer overflow. Il valore iniziale del canarino è solitamente casuale o dipendente dal programma in esecuzione (altrimenti sarebbe possibile sfondare i limiti dei buffer riconoscendo e saltando i canarini).

Inoltre non sempre le convenzioni sugli indirizzi delle variabili permettono questo tipo di attacco.

SQL Injection

Come abbiamo visto un attacco buffer overflow è possibile nel momento in cui i dati in input sono deliberatamente mal formati dall'attaccante in modo da ottenere un risultato non previsto dal programmatore.

Un'altro tipo di attacco basato sullo stesso principio è **SQL injection**, uno degli attacchi ai siti web più diffuso al mondo.

Storia recente

A differenza dell'attacco **buffer overflow**, gli attacchi di tipo SQL injection sono tuttora molto diffusi.

Anonymous (il team di hacker recentemente balzato all'attenzione dei media per il caso Wikileaks), non più tardi di Febbraio 2011 è penetrato nel sistema informativo della HBGary (la società di sicurezza informatica consulente del governo USA) proprio grazie ad un attacco di questo tipo.

Esempio

Supponiamo di essere i gestori di un sito web a registrazione. Ogni utente può sottoscrivere un abbonamento specificando un indirizzo e-mail, uno user name e una password. Il database relazionale del nostro sito conterrà quindi una tabella di questo tipo:

id	**username**	**email**	**password**
1	julian	julian@nowhere.com	67c2c13e9cc0 ...
2	gdavid	gilmour@music.co.uk	297aae72cc4 ...
3	topolin	mickey@disney.com	098f6bcd462 ...
...			

Table: La tabella *tblUsers*

Esempio, cont'd

Tipicamente i siti come questo forniscono una pagina di **lost password** nel caso di smarrimento dei dati di accesso.

Inserendo il proprio username, in un apposito modulo, l'utente riceve un messaggio di posta elettronica con le istruzioni per la generazione di una nuova password, o addirittura la password stessa.

Esempio, cont'd

Supponiamo che l'utente `gdavid` abbia smarrito la password e venga quindi rediretto sulla pagina del servizio di reinvio password. Il sistema dovrà richiedere al database, tramite una query SQL, l'indirizzo e-mail dell'utente e la password fornita al momento della registrazione.

Esempio: in una pagina scritta in PHP, se la variabile `$uname` contiene il nome utente specificato nel modulo, la query verrà composta per sostituzione in questo modo:

```
SELECT email FROM tblUsers WHERE username = "$uname";
```

Esempio, cont'd

Se non viene fatto alcun controllo sui dati (pratica molto comune), e quindi sul valore della variabile `$uname`, è possibile inserire codice SQL al posto del nome utente, ad esempio `whatever"; TRUNCATE TABLE tblUsers - -`

Poichè la query viene costruita per sostituzione, il comando che verrà inviato al server SQL sarà:

```
SELECT email FROM tblUsers WHERE username =
"whatever"; TRUNCATE TABLE tblUsers - - "
```

Esempio, cont'd

`TRUNCATE TABLE tblUsers` è il codice SQL per lo svuotamento totale di una tabella, e `;` è, in SQL, l'operatore per concatenare due query.

`- -` indica un commento nella sintassi SQL, e serve all'attaccante ad evitare che le virgolette finali causino un errore di sintassi impedendo alla query di essere eseguita. Il risultato è che la tabella `tblUsers` viene cancellata da un utente qualsiasi del sito (neppure necessariamente autenticato).

Perché succede?

Il problema è che alcuni dei caratteri utilizzabili nei moduli su Internet sono anche presenti nella sintassi SQL e poichè spesso i dati di input vengono sostituiti all'interno delle query SQL, è possibile modificare il significato delle query.

Vi sono diverse soluzioni a questo problema.

Contromisure

- **Escaping** consiste nel convertire ogni carattere in una sua rappresentazione testuale che non abbia alcun significato in SQL. Per esempio, al posto delle doppie virgolette " viene messo il carattere \". Per semplificare queste procedure i linguaggi di programmazione di alto livello forniscono funzioni di conversione apposite (e.g., in PHP `mysql_real_escape_string()`),
- **Disable query concatenation** per evitare attacchi come quello appena visto alcuni linguaggi impediscono di concatenare le query. In questo modo non sarà possibile, ad esempio, appendere una `TRUNCATE` ad una `SELECT`, però non viene garantita l'integrità della `SELECT` stessa.

SQL injection nella cultura popolare

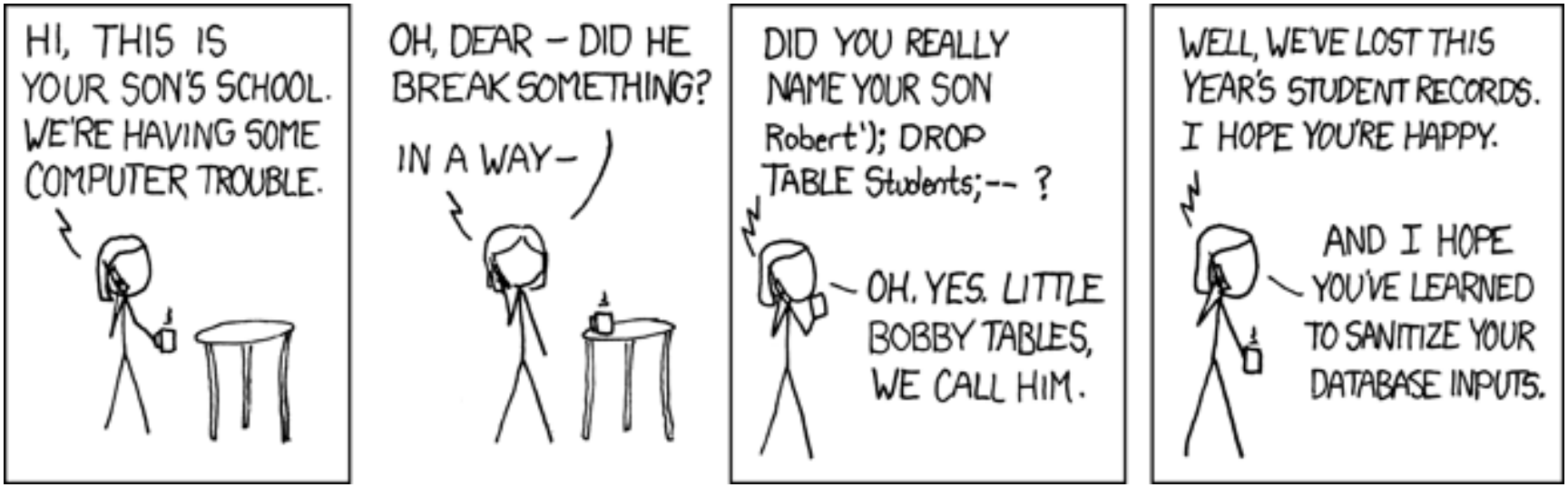


Figure: *Exploits of a Mom* - http://xkcd.com

Conclusioni

I dati in input possono generare famiglie di attacchi molto difficili da prevedere e individuare, per questo motivo nessun input può essere considerato "sicuro" a priori. Ogni dato proveniente dagli utenti dovrebbe essere controllato e validato, soprattutto se il software è esposto ad accessi dalla rete.

Esercizi su buffer overflow

Su **buffer overflow**:

- Compilare `vulnerable.c` ed `exploit.c` usando il Makefile.
- Eseguire `vulnerable` input di varia lunghezza. Eseguire `exploit`.
- Provare a compilare `vulnerable.c` senza l'opzione `-fno-stack-protector`, cosa succede? Perché?
- Provare a compilare `vulnerable.c` senza l'opzione `-z executestack`, cosa succede? Perché?
- Provare a compilare `generator.c` ed eseguire `./generator comando` (e.g. comando = `ls -AGal`).
- Utilizzare la stringa in output da `generator` come shellcode in `exploit.c`.

Esercizi su SQL injection

Su **SQL injection**:

- Provare a ottenere lo svuotamento della tabella.
- Reinserire qualche utente dal modulo apposito.
- Provare ad ottenere l'inserimento di un utente senza passare dal modulo apposito (e.g. evitando un controllo sulla validità dell'indirizzo e-mail).
- (Avanzato) c'è un altro oggetto della pagina a rischio SQL injection, quale? Provare a svuotare la tabella sfruttando questa vulnerabilità.
- (Avanzato) ottenere l'eliminazione (non svuotamento) della tabella.

All'indirizzo **http://localhost/sqlinjection** sulle vostre macchine è disponibile un playground dove potete provare gli esercizi.

Riferimenti


MicroSoft | TechNet
The 10 Immutable Laws of Security.
`http://technet.microsoft.com/en-us/library/cc722487.aspx`


Elias Levy (aka AlephOne)
Smashing the Stack for Fun and Profit, 1996.
`http://insecure.org/stf/smashstack.html`


Andrea Cugliari, Mariano Graziano, 2010
Smashing the Stack in 2010, 1996. `http://5d4a.wordpress.com/2010/08/02/smashing-the-stack-in-2010`


Wikipedia
SQL Injection `http://en.wikipedia.org/wiki/SQL_injection`


Wikipedia
Buffer overflow `http://en.wikipedia.org/wiki/Buffer_overflow`

fine