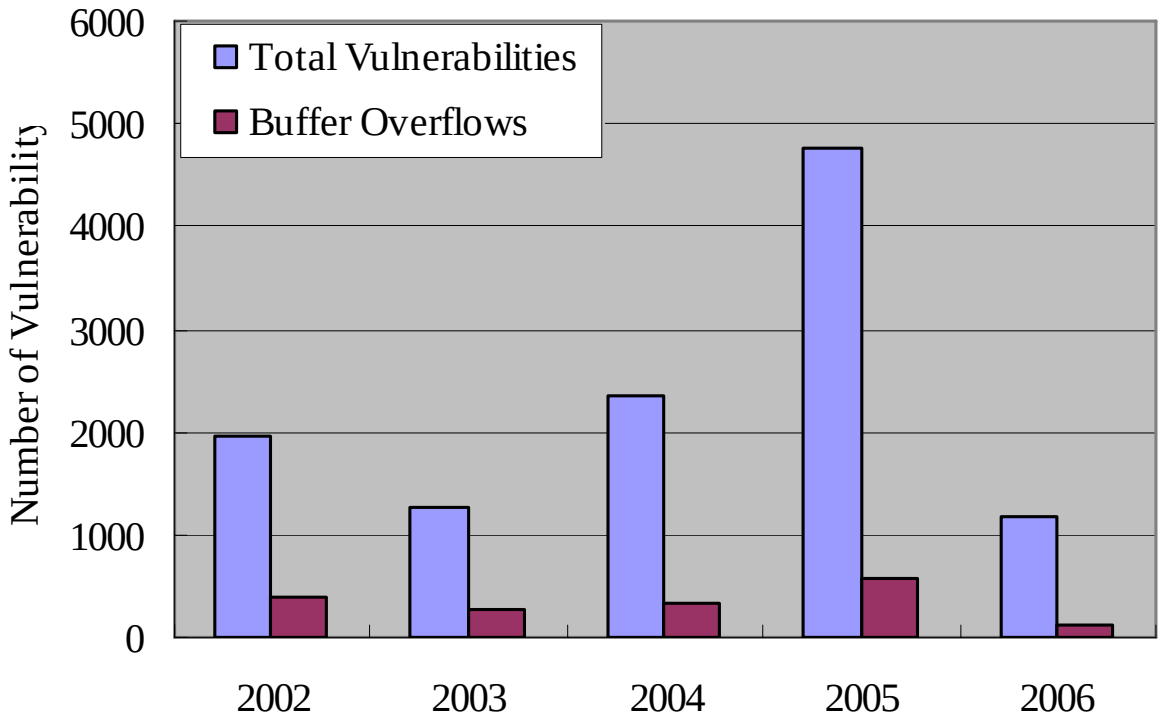
LibsafeXP: A Practical & Transparent Tool for Run-time Buffer Overflow Preventions

Zhiqiang Lin, Bing Mao and Li Xie
Dept. of Computer Science
Nanjing University, P.R.China
June 23, 2006

Agenda

- Background
- Our Approach
- Evaluation
- Discussion
- Related Work
- Conclusion

Buffer Overflow: Common



Buffer overflows in NIST National Vulnerability Database

Buffer Overflow: Severe

- Inject malicious code
- Overwrite program critical data structures
- Execute Attacker's malicious code
- ...
- Worms
  - Code Red, SQL Slammer, Blaster, etc.

Limitations of Previous Approaches

- Access to program source code
  - e.g., StackGuard, CRED
- Significant performance overheads
  - e.g., J&K
- Require hardware support
  - e.g., SmashGuard
- Require debugging information
  - e.g., LibsafePlus
- ...

- Thus, it is still necessary to provide both *practical* and highly *efficient* solution to prevent buffer overflows.

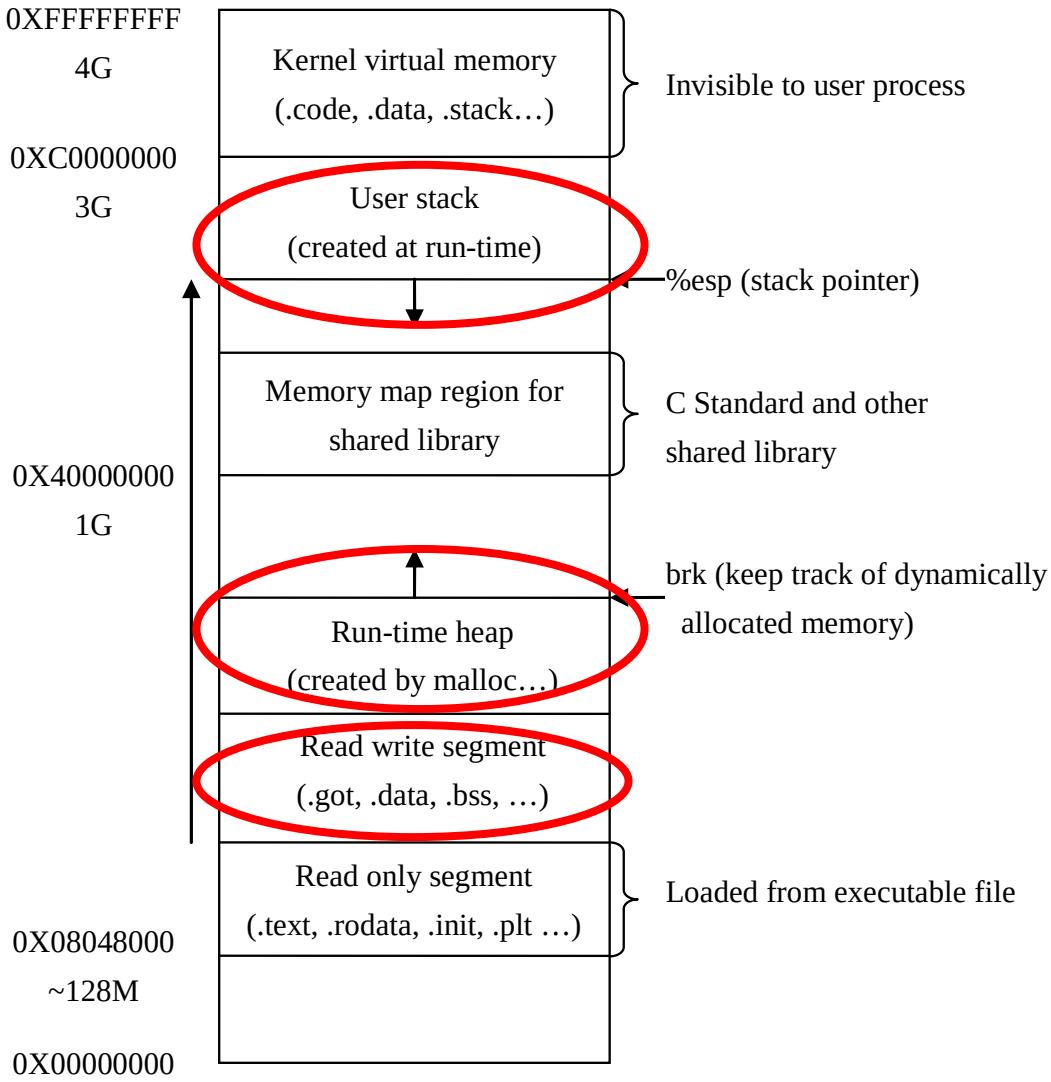
Our Approach: A lightweight tool ,LibsafeXP

- Add bounds checking for all the program dereferencing buffer.
  - Global Buffers
    - Its size and starting address are extracted from the symbol table section of ELF
  - Heap Buffers
    - Tracked at run-time in the intercepted malloc family functions.
  - Stack Buffers
    - Frame pointer, as Libsafe to calculate.

Advantages:

- Practical in application.
- Effective against buffer overflow attacks.
- Easy to use.
- Low run-time overheads.

Memory Layout of UNIX Process



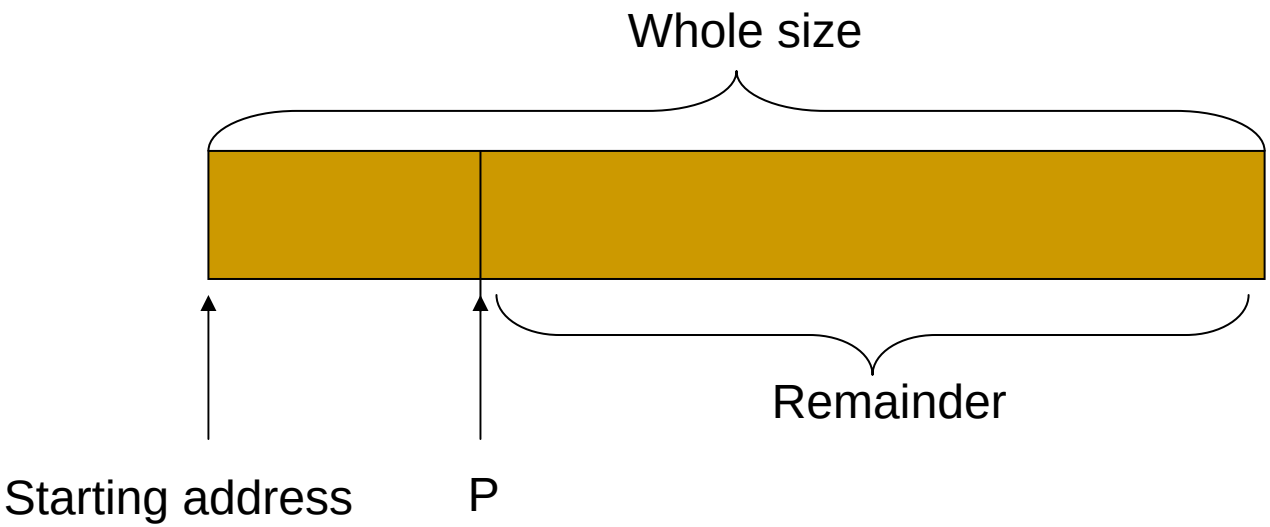
An Example

```
1 #include <stdio.h>
2 #include <string.h>
3 char str1[10]; //in .bss
4 char str2[20]="a test string"; //in .data
5 int main() {
6     static char str3[30]; //in .bss
7     static char str4[30]="in .data";
8     strcpy(str1,str2);
9     strcpy(str3,str4);
10    printf("str1=%s\n",str2);
11    return 0;
12 }
```

Whole size. How to address the members of record variable?

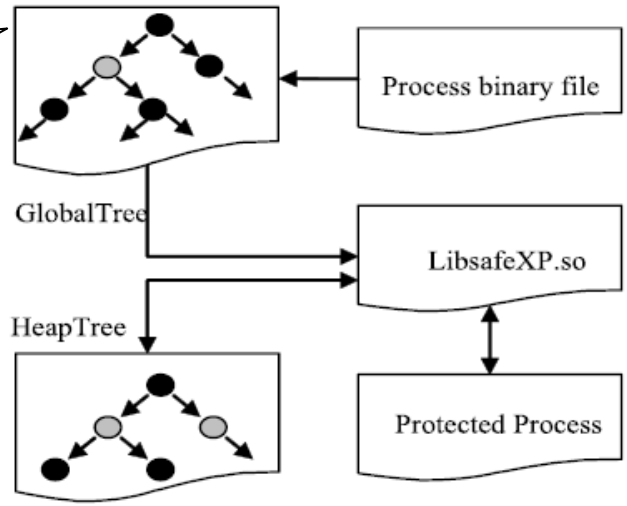
```
...
08049714   30   OBJECT  LOCAL   .bss    str3.3
080495dc   30   OBJECT  LOCAL   .data   str4.4
08049734   10   OBJECT  GLOBAL  .bss    str1
080495c8   20   OBJECT  GLOBAL  .data   str2
...
080483b4   91   FUNC    GLOBAL  .code   main
...
```

Remainder of the whole size



Overview of LibsafeXP

```
struct t_tree_node{
        enum {red, black} colour;
        char *addr;
        int size;
        struct t_tree_node *left,
                           *right,
                           *parent;
}
```



Buffer Overflow Prevention in LibsafeXP

- In our approach, for any dereferencing destination buffer α, its limited maximum access range f(α) is calculated by

$$f(\alpha) = \begin{cases} EBP - \alpha & \text{if } \alpha \in Stack \\ T(\alpha).size - (\alpha - T(\alpha).addr) & \text{if } \alpha \in GlobalTree \cup HeapTree \\ 0 & \text{if } \alpha \notin Stack \cup GlobalTree \cup HeapTree \end{cases}$$

where *EBP* is the pointer to the stack frame in which α resides, and $T(\alpha)$ is the most nearest node that could contain address α in our red-black tree.

Bounds checking on global buffers

```
...
struct test {
        int a[600];
        char buf[20];
}A;
char p[20];
char str2[]="hello world\n";
....
foo(){
        ...
        strcpy(p+4,"buffers...");
        strcpy(A.buf,str2);
        ...
}
```

$$f(p+4) = T(p+4).size-((p+4)-T(p+4).addr)$$
$$= 20- ((p + 4) - p) = 16$$

$$f(A.buf) = T(A.buf).size - ((A.buf) - T(A.buf).addr)$$
$$= 620 - (A.buf - A)$$
$$= 620 - ((A + 600) - A) = 20$$

Bounds checking on heap buffers

- Once these malloc family functions are called, add the newly allocated symbol's starting address and associated size into our *HT*.

- And use the same buffer overflow prevention method described above to determine the legal access range so as to defend against heap buffer overflows.

Bounds checking on stack buffers

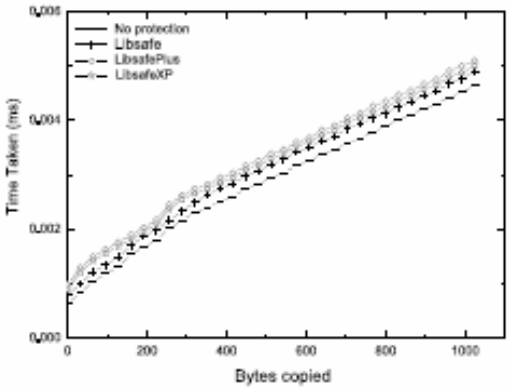
- Local variable's symbol information is not available in the program symbol tables. When the wrapper functions refer these local buffers, they cannot find the relevant address and size information.

- Fortunately, based on the fact that once overflow occurs local buffers would smash the frame pointer, we can hence use the saved frame pointer as Libsafe [2] did to act as the upper bounds when program writes to destination address in stack.

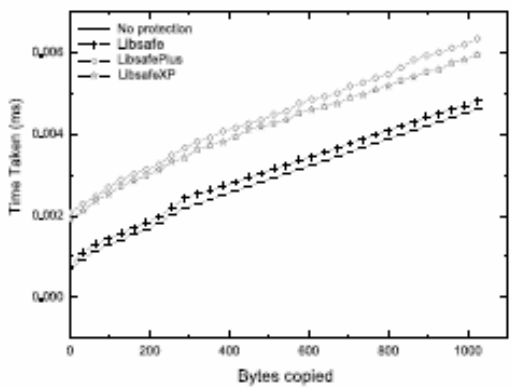
Effectiveness

- Protection against stack buffers
  - no worse than Libsafe
- Protection against global and heap buffers
  - All the attack techniques developed in Wilander's test suite attempting to overwrite program .data and .bss global variables were successfully detected and prevented.
  - For the .heap buffer overflows, also as expected, all the out-of-bounds write were successfully caught and prevented.

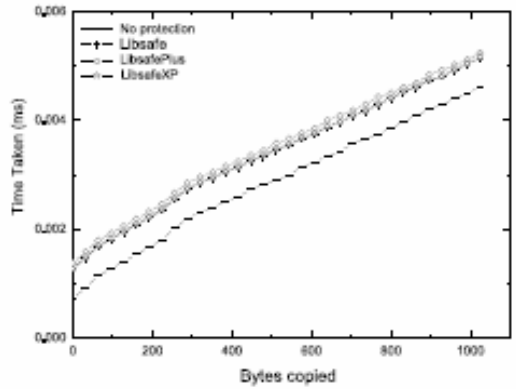
Micro-benchmark



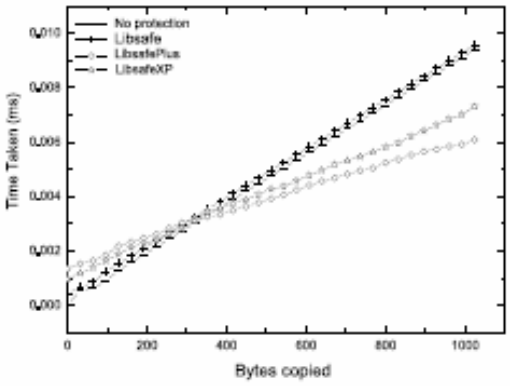
(a) memcpy() to a global buffer



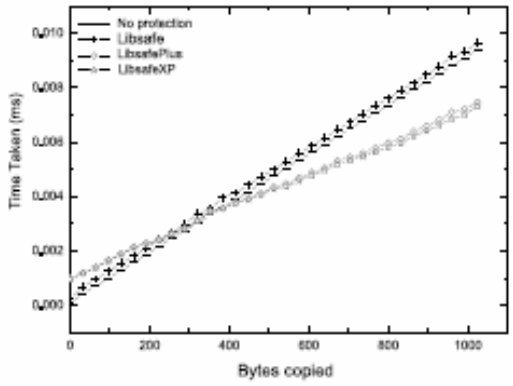
(b) memcpy() to a heap buffer



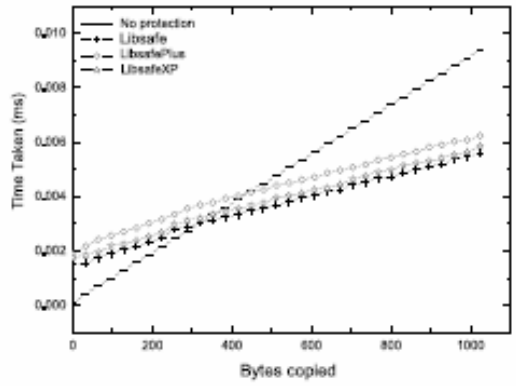
(c) memcpy() to a local buffer



(d) strcpy() to a global buffer



(e) strcpy() to a heap buffer



(f) strcpy() to a local buffer

Macro-benchmark

Applications	Libsafe	LibsafePlus	LibsafeXP
Apache-2.0.48	1.0X	1.0X	1.0X
Bison-1.875	1.2X	2.4X	2.1X
Enscript-1.6.1	1.1X	1.3X	1.3X
Hypermail-2.1.8	1.0X	1.1X	1.1X
OpenSSH-3.7.1	1.0X	1.0X	1.0X
OpenSSL-0.9.7	1.0X	1.0X	1.0X
Gnupg-1.2.3	1.0X	1.0X	1.0X
Grep-2.5	1.1X	1.3X	1.3X
Monkey-0.8.5-1	1.1X	1.3X	1.3X
Ccrypt-1.2-2	1.0X	1.0X	1.0X
Tar-1.13.25-4	1.0X	1.0X	1.0X

Limitations

- False negatives

```
struct some_global_struct{
        ...
        int (*foo_a)();
        char buf[N];
        ...
        int (*foo_b)();
        ...
        int (*foo_c)();
        ...
};
int (*foo_d)();
```

Limitations

- False negatives
- Symbol table
- Standard C Library function
- Dynamic Link

Related Work

- Static analysis
- Compiler extensions
- Safe library functions
- Execution monitoring
- Intrusion detections
- Randomizing code/space transformations
- ...

Libsafe & Libverify

- Libsafe and Libverify
  - Libsafe provides secure calls to the buffer re-lated glibc functions
  - Libverify uses a similar approach to StackGuard by verifying the function return address before use.
  - Both of the two approaches were powerful, but they only focused on the prevention of stack buffers.

LibsafePlus

- TIED and LibsafePlus
  - TIED extracts buffer's size from program debugging information which contains not only the global buffer's size but also the local variables, to help
  - LibsafePlus determine buffer overflows.

  - LibsafePlus is a very promising approach for buffer overflow preventions, but it heavily relies on the program debugging information.

LibsafeXP

- Since
  - (i) LibsafePlus requires the debugging information, which is usually unavailable in the released software
  - (ii) Libsafe only provides limited scope checking
- We extend and integrate them to implement our tool LibsafeXP.

- Although LibsafeXP looks like LibsafePlus, they are based on different knowledge.
  - Debugging Section
  - Symbol Section

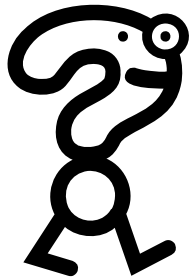
Conclusion

- A practical tool, LibsafeXP, to guard against almost all the three types of buffer overflows dynamically and transparently.
  - For global variables, we rely on the symbol section of the protected ELF executable file, and extract those information for our bounds checking.
  - For heap-based buffers, we intercept those related buffer operation APIs and track the allocated buffer's size.
  - For stack-based local variables, as there is no such information available, we use the frame pointer as the upper bound.

Future Work

- Extend LibsafeXP to other platforms, such as Windows for PE files.

Q & A



- linzq@dislab.nju.edu.cn
- maobing@nju.edu.cn
- xieli@nju.edu.cn

- Thank you