

Security Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2015

Outline

- Introduction and applications
- Buffer overrun vulnerability

Introduction

- Security in a broad sense.
 - Effects: crash, non-termination, wrong output, unintended actions
 - Causes: dangling pointers, buffer overruns, null pointer dereference, wrong opcode, arbitrary data-change
- C programs are more susceptible to buffer overflow attacks.
- C allows direct pointer manipulation – since space and performance are primary concerns – not security.
- Standard library contains functions that are unsafe if not used carefully (e.g., *gets*, *strcpy*, *strcat*). Does *str***n***cpy* solve the problem?

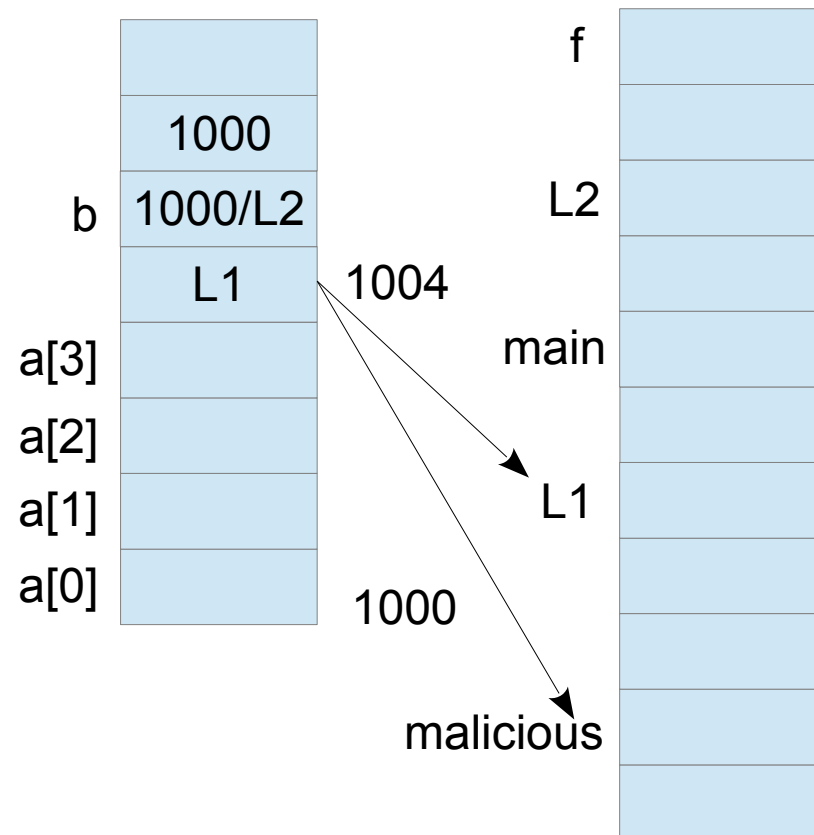
Stack Smashing

- How can a malicious code be executed by exploiting buffer overrun vulnerability?

```
void f(char *b) {
    gets(b);
L2:
}
void main() {
    char a[4];
    f(a);
L1: ...
}
```

```
f:
    pop b
    push L2
    push b
    jump gets
    ...
    pop PC

main:
    mov a, SP
    add SP, 4
    push L1
    push a
    jump f
```

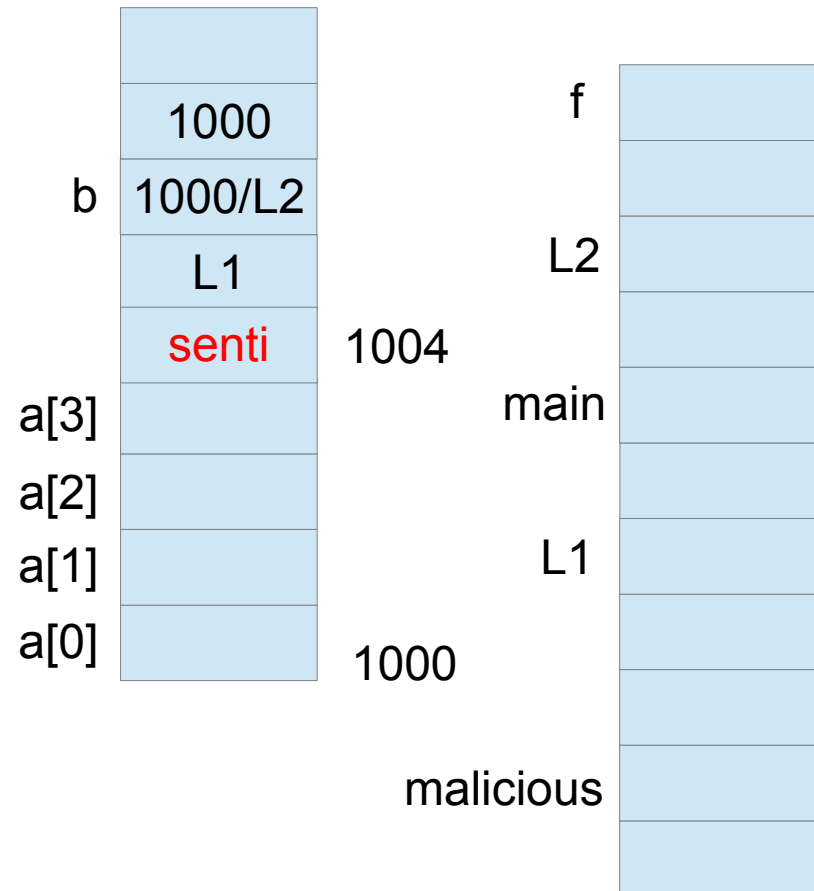


To Avoid Stack Smashing

- Insert a sentinel near the return address.
- Check if it is intact before jumping.

```
void f(char *b) {  
    gets(b);  
L2:  
}  
void main() {  
    char a[4];  
    f(a);  
L1: ...  
}
```

```
f:  
    pop b  
    push senti  
    push L2  
    push b  
    jump gets  
    ...  
    intact senti?  
    pop PC  
  
main:  
    mov a, SP  
    add SP, 4  
    push senti  
    push L1  
    push a  
    intact senti?  
    jump f
```



To Avoid Stack Smashing

- Insert sentinel / canary
- Check addresses / bounds explicitly (Java)
- Wrap system calls with security checks

Dynamic techniques

- Runtime overhead
- Program is terminated

- When the code segment is writable, it is more vulnerable to attacks (*self-modifying code*, W^X).
- What does the following program do?

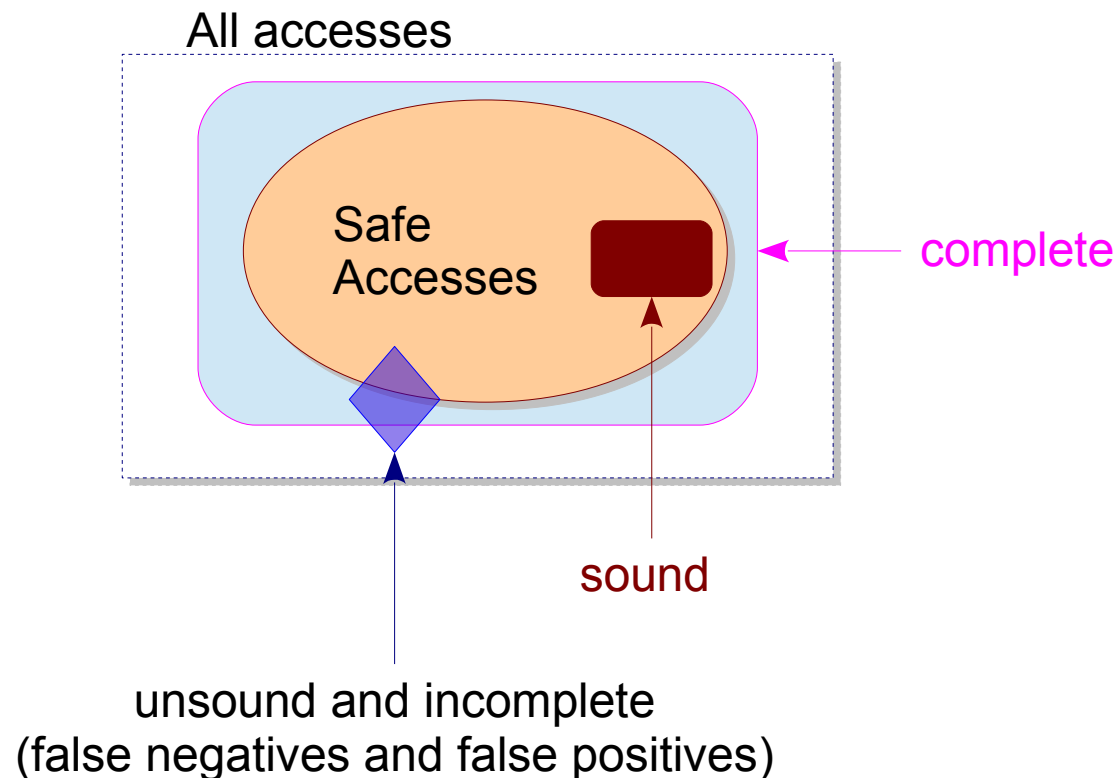
```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}
```

Notes on Stack Smashing

- Using canary for stack smashing detection?
 - Canary is a bird used in coal-mines to detect toxic gases (humans follow the caged birds)
 - Researchers have validated its performance impact to be minimal
 - Randomizing canary improves odds
 - Does not *guarantee* protection
- How about heap smashing?
 - Heap usually doesn't contain return addresses
 - But then, we have function pointers

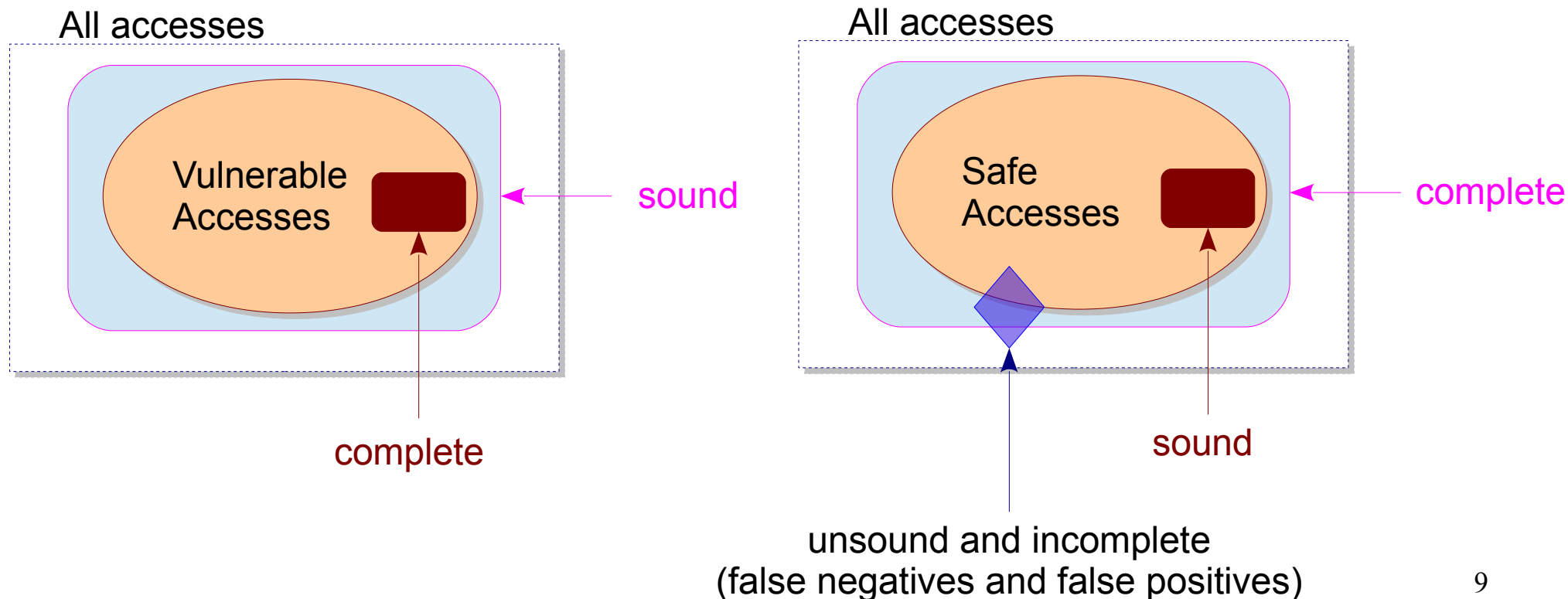
Static Buffer Overrun Detection

- A good example of static analysis that can be incomplete as well as unsound.



Static Buffer Overrun Detection

- A good example of static analysis that can be incomplete as well as unsound.



Using Pre and Post-conditions

- Annotations define properties
 - *minDef, maxDef, minUse, maxUse*
e.g., `minDef(buff) = 0, maxUse(buff) = N / 2`
 - *notNull, null, restrict*
e.g., `notNull(ptr), restrict(ptr)`
 - **Homework:** Write an example program using `restrict` which enables an optimized code.
- Initially we would assume that these annotations are user-provided. Later, we will try to auto-infer them.

Specifying Pre and Post-conditions

- **char *strcpy(char *s1, char *s2)**
/* @requires maxDef(s1) >= maxDef(s2) */
/* @ensures maxUse(s1) == maxUse(s2)
 and result == s1 */;
- **void *malloc(size_t size)**
/* @ensures maxDef(result) == size
 or result == null */;

Inferring Constraints

- From the **for**-loops init, bound and change
 - Difficult for general loops such as **while**
- From the array declarations and malloc statements
- From conditional checks in the code
- Small number of heuristics often cover large part of the program.
- Once the constraints are identified, these are checked against the user annotations.

Inferring Constraints

- In absence of annotations, simply generating all possible constraints is expensive.
- In the past, researchers have tried flow-insensitive constraints.
- Auto-inference is feasible when loop-bounds do not depend on **values**.
 - **while** (a[i] != '\0') **versus** **while** (i < n)

Precision vs. Efficiency

```
void main() {  
    int *a;  
    a = malloc(N);  
    ii = N / 2 + f(N);  
    a[ii] = 0;  
}
```

```
...  
int f(int N) {  
    return N % 5;  
}
```

- Precision requires interprocedural analysis in the above example (recall Analysis Dimensions).
- Domain knowledge about N may help in filtering out false positives.

Stack Smashing in gcc

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
        printf ("\n Wrong Password \n");
    else
        printf ("\n Correct Password \n"),    pass = 1;

    if(pass)
        /* Now Give root or admin rights to user*/
        printf ("\n Root privileges given to the user \n");

    return 0;
}
```

Source: Ramesh Natarajan, thegeekstuff.com

Older gcc

Enter the password :
hhhhhhhhhhhhhhhhhhhhhh

Wrong Password

Root privileges given to the user

New gcc

Enter the password :
hhhhhhhhhhhhhhhhhhhhhh

Wrong Password

*** **stack smashing detected** ***: ./a.out terminated

New gcc with *-fno-stack-protector*

Enter the password :
hhhhhhhhhhhhhhhhhhhhhh

Wrong Password

Root privileges given to the user

Vulnerability Analysis as a DFA

- Data-flow facts
- Statements of interest
- Analysis direction
- Meet operator



Classwork

Vulnerability Analysis in Polyhedral Model

- How do you model inequalities?
- What are the constants?
- What do you get after solving the system?

Tools

3. BOON

- Array out of bound check for C
- Flow-insensitive, intra-procedural, pointer-insensitive

2. CQual

- Annotation-based
- Uses type qualifiers to propagate taint annotation
- Detects format string vulnerability by type checking

Tools

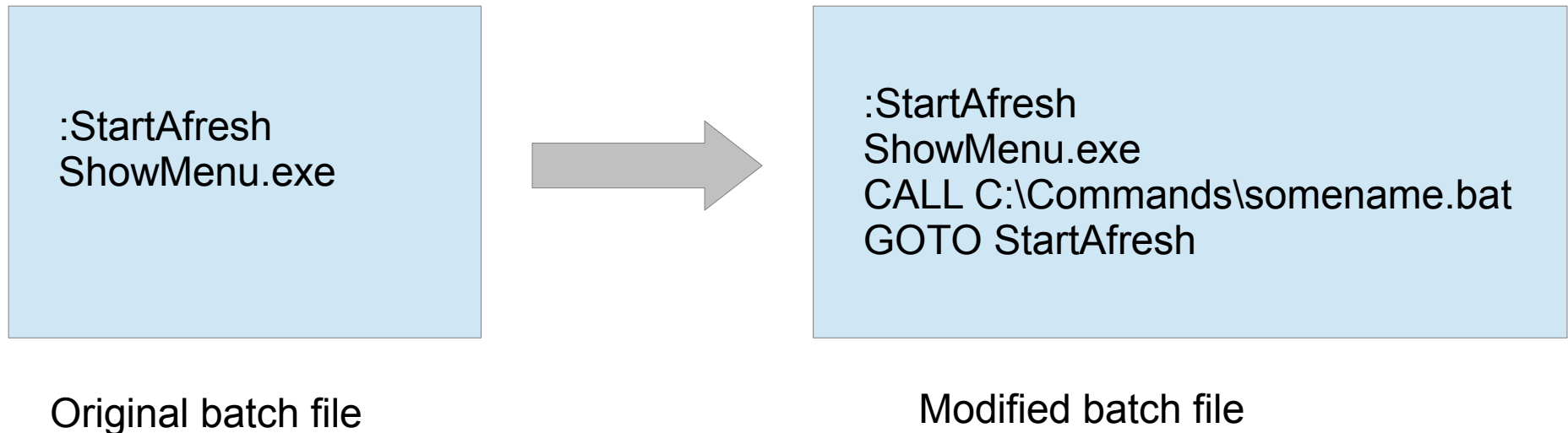
1. `xg++`

- Template-driven compiler extension
- Finds kernel vulnerabilities
- Tracks kernel data originated in untrusted source, memory leaks, deadlock situations

0. Eau Claire

- Theorem-prover based (specification-checker)
- Finds buffer overruns, file access races, format string bugs

Self-Modifying Code



In earlier single-window DOS systems, only one window could be active, and easy inter-process communication was not well-developed.

TCF

- Program Analysis CS6843
- Rupesh Nasre 008606
- G1..G5
- (i) 102 and 111
- (ii) 201 and 206
- (iii) 303 and 311
- (iv) 407 and 411
- (v) 506 and 508
- (vi) 601