Applied Cryptography and Network Security

Adam J. Lee

adamlee@cs.pitt.edu

6111 Sennott Square

Lecture #23: OS and Application Security

April 3, 2014

University of Pittsburgh

Overview

OS protection model overview

- Memory protection
- Transitions between user mode and kernel mode

Integrity of mechanism

Compromising integrity of mechanism by violating assumptions at

- The OS level: Buffer overflow case study
- The application level: SQL injection case study

Protection in x86 Processors

x86 processors operate in two distinct modes

- Kernel mode (privileged)
- User mode (unprivileged)

The kernel is typically "small" when compared to the entire OS

- Controls base system functionality
- Provides interface to hardware
- Manages system data structures

Why use a small trusted kernel?

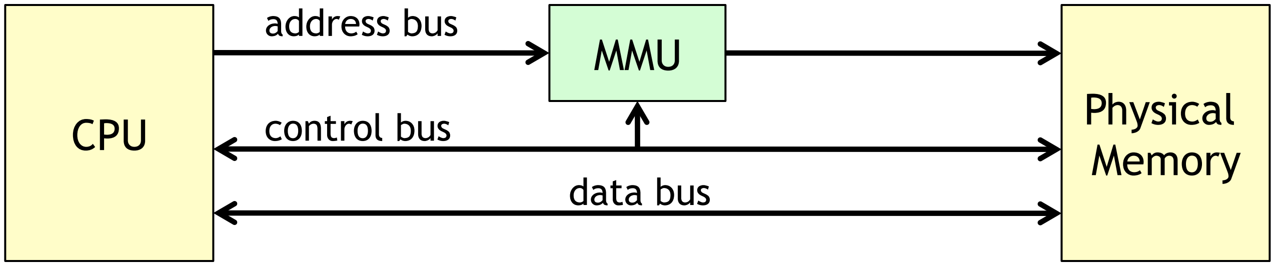
- In theory, a small kernel can be rigorously tested and verified
- In practice, it's always good to minimize your trusted computing base
    - Recall Saltzer and Schroeder's principle of least privilege

One critical job of the OS is memory protection

A user should be unable to access memory used by the kernel or other users

- This provides isolation, which is necessary to ensure integrity

How is this enforced?



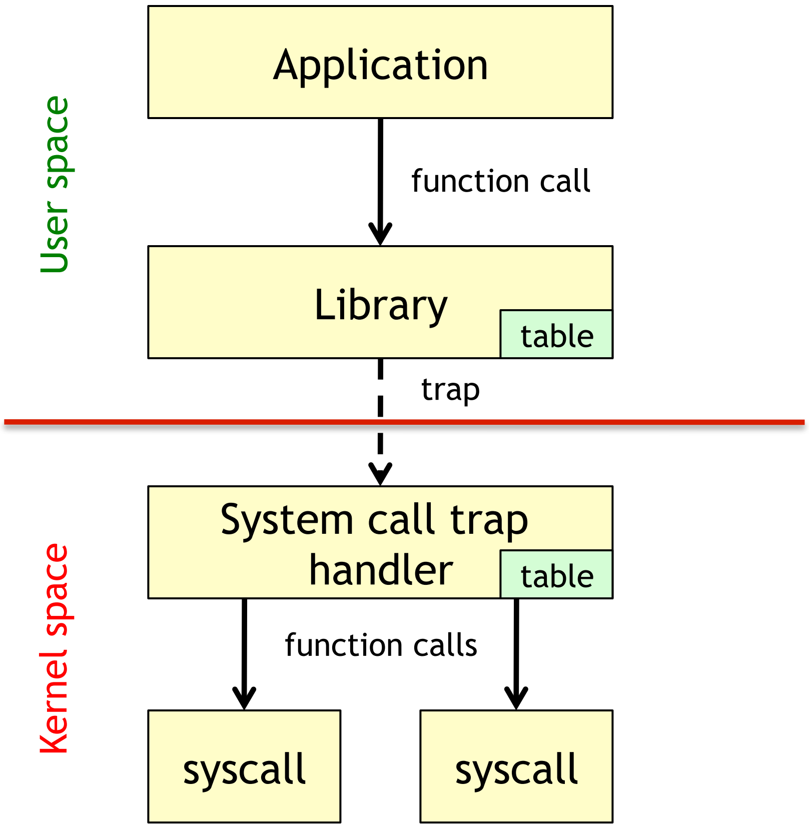
The memory management unit (MMU) plays the role of the enforcer

- Translates logical memory addresses into physical addresses
- Allows regions of memory to be marked as read only
- Ensures that requests are a valid
  - Point to valid physical addresses
  - Are permissible (e.g., not writing to read only memory)

The OS controls the MMU and manages its lookup tables

It is unreasonable to believe that we can completely separate users from the kernel

System calls allow user space code to access OS services



System calls transfer control of the CPU from the user to the kernel

How does this process work?

- User code identifies which kernel service it wants to invoke (e.g., by leaving a code in a specific register)
- Arguments for this service are left on the program stack
- Trap instruction issued
- Trap handler looks up code and gathers any arguments from the stack
- A function call is made

Recall from your OS class that the above process is called a context switch

OS security depends on integrity of mechanism

The protection mechanisms used by the OS are designed to narrow the interface through which users can access protected data

In order to ensure that the system remains secure, we need to ensure that

- The principle of complete mediation is followed
- “Trusted” code is actually trusted

There is also an implicit assumption that user processes behave as expected by the users who invoke them

As a result, compromising the security of an OS typically involves violating the integrity of its enforcement mechanisms

How can we violate the OS's assumption of integrity of mechanism?

Trivial attack: Extend the TCB by installing a malicious device driver

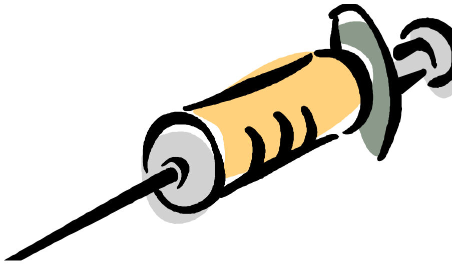
- E.g., Why not write a keyboard driver that's also keystroke logger?
- Drivers are trusted, and have complete access to kernel data structures
- This is a viable attack method, but its uses are limited

A better idea is to make an existing trusted process behave badly

Approach 1: Kernel Level

Use a buffer overflow to subvert the kernel's context switch process

Approach 2: Application Level



Violate application-level constraints via SQL injection

Buffer overflow is one of the oldest tricks in the book

In fact, buffer overflow was one of the attack vectors exploited by the Morris worm back in 1988!

We'll learn more about the Morris worm in later lectures

Informally, a buffer overflow works by providing an extremely long input to a user program that causes system data structures to be overwritten

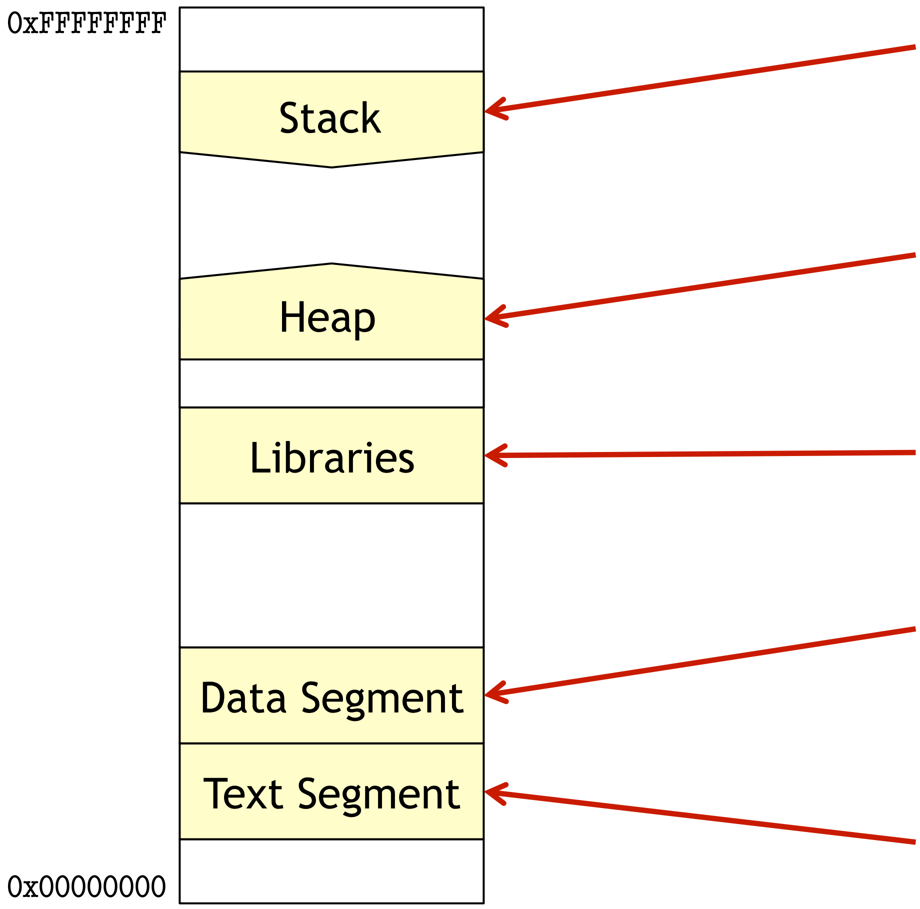
The result of this is that control is transferred to code injected by the attacker, instead of being transferred back to the parent process

This attack vector is best used against root-level processes

- Why? The attacker's code will run as root!
- This provides a way for the attacker to "trick" the system into granting him privileged access

Understanding how buffer overflows work means understanding system memory management

Usually, a process's address space is a chunk of virtual memory accessed as an array of bytes



The stack is used by processes for book keeping purposes. The stack grows downward in the address space

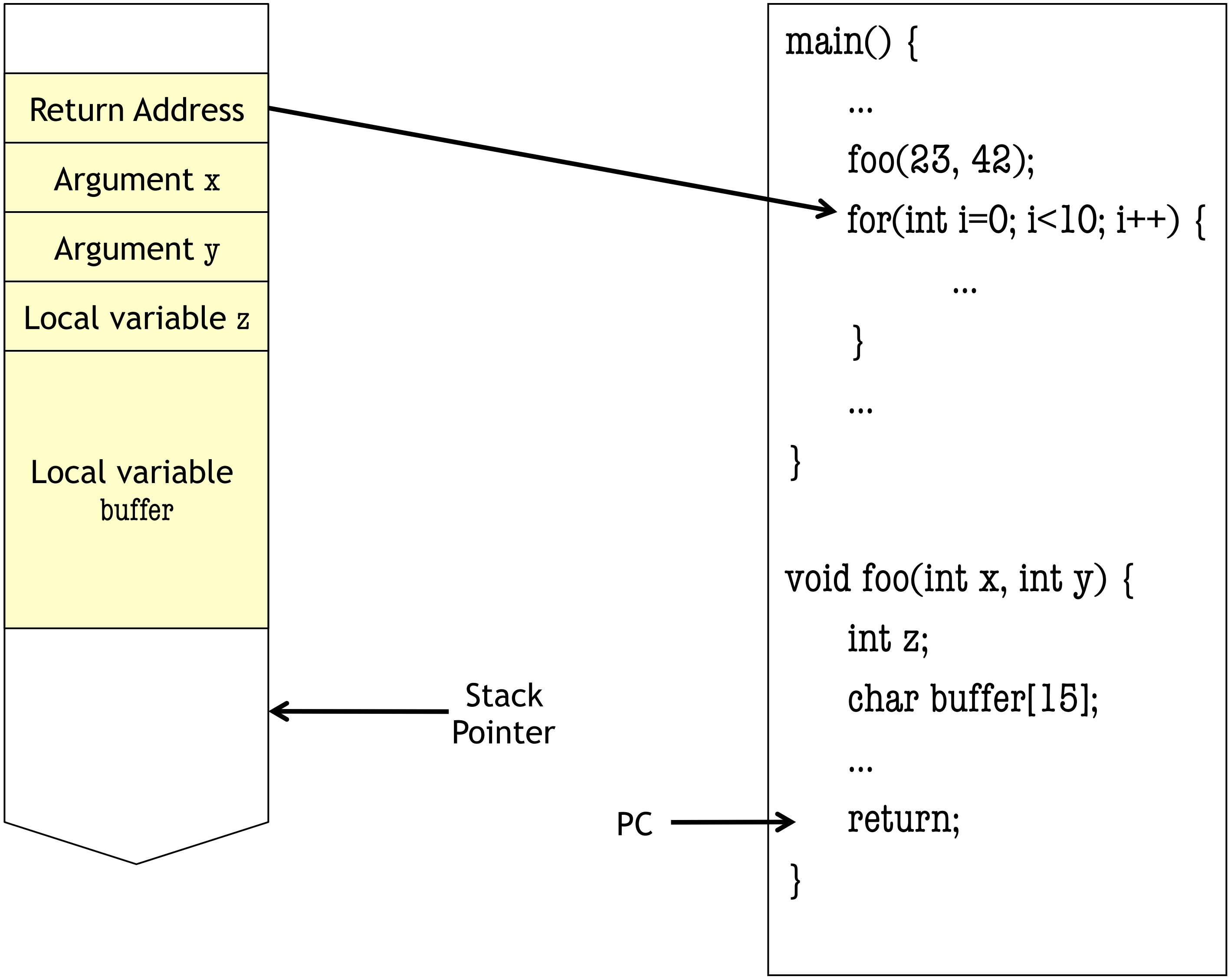
Dynamically created data structures are allocated at runtime on the heap. The heap grows upward in the address space

Library functions used by the programmer are linked into the address space, often times at well-known addresses

Global variables and static data structures live here

The set of instructions for a program are fetched into the text segment

How does the stack work?



```
main() {
    ...
    foo(23, 42);
    for(int i=0; i<10; i++) {
        ...
    }
    ...
}

void foo(int x, int y) {
    int z;
    char buffer[15];
    ...
    return;
}
```

Smashing the stack

Aleph One, Smashing the Stack for Fun and Profit, Phrack Issue 49, Nov. 1996.

In a stack smashing attack, the attacker's goal is to:

- Inject exploit code onto the stack
- Overwrite the return address pointer to transfer control into the exploit code, rather than the calling function

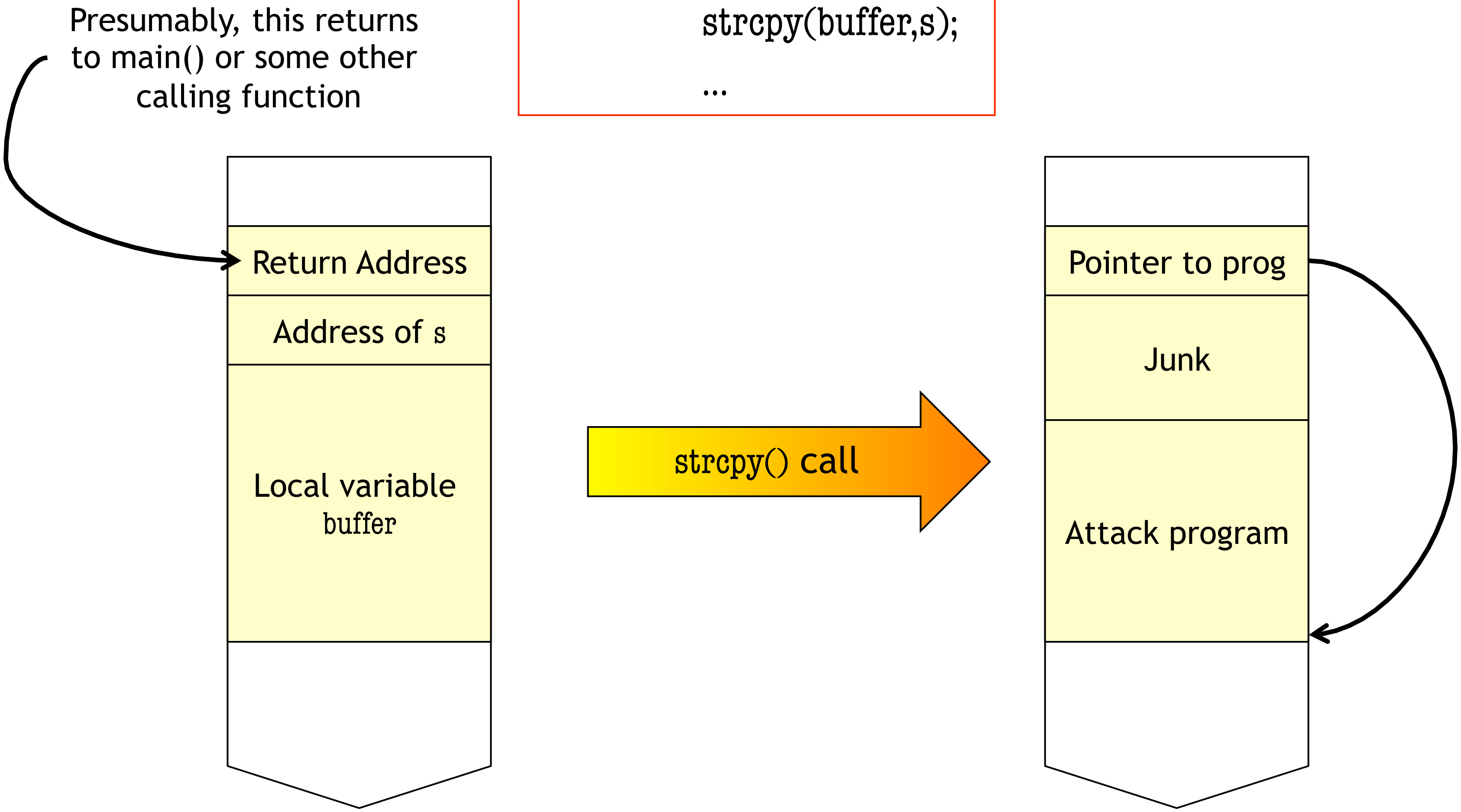
Result: If the exploited code was running as an administrator the attacker's code executes with administrative privileges!

- Unix servers often run as root
- Most Windows users log in with administrative privileges

This can often be accomplished using unsafe string operators in C

For example, consider the following program fragment

```
void bar(char *s) {
        char buffer[100];
        strcpy(buffer,s);
        …
```



How can we craft an exploit for the overflow?

Often our exploit code is just a few system calls

Caveat: Writing good shellcode is a bit of an art

- The attacker needs a good understanding of memory layout
- Code needs to be location independent
- If the code is exploiting C string functions, it cannot include any 0x00 bytes

Assuming that we can do that (see MetaSploit project) we must still answer two critical questions:

1. Where is the buffer and how big is it?
2. Where is the return address?

In reality, we only need to have approximate answers to these questions!

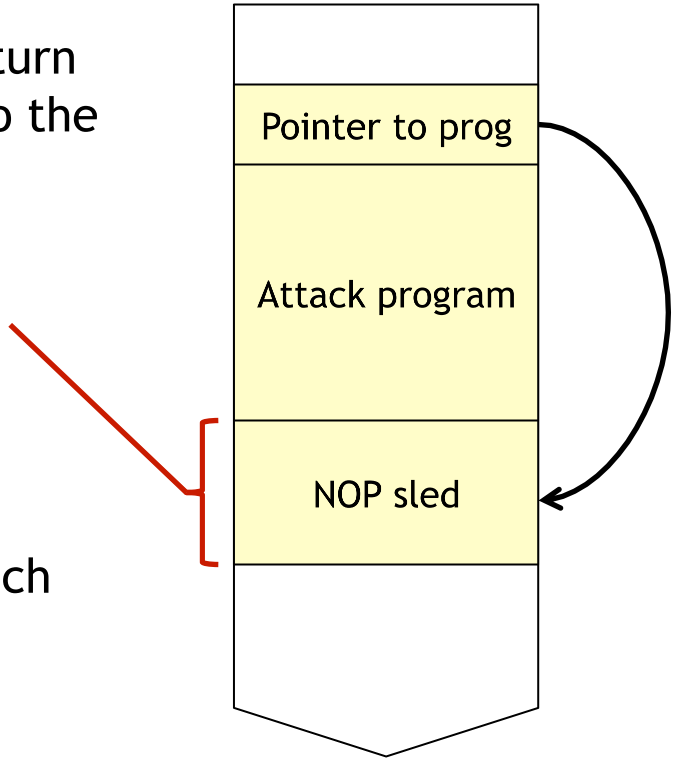
NOP sleds help us guess the program’s start location on the stack

Most CPUs support a *No Operation* instruction of some sort

Rather than starting our attack program at the beginning of our buffer, we can pad it out with a bunch of NOP instructions

This is good, because the overwritten return address does not need to point exactly to the location of our exploit code!

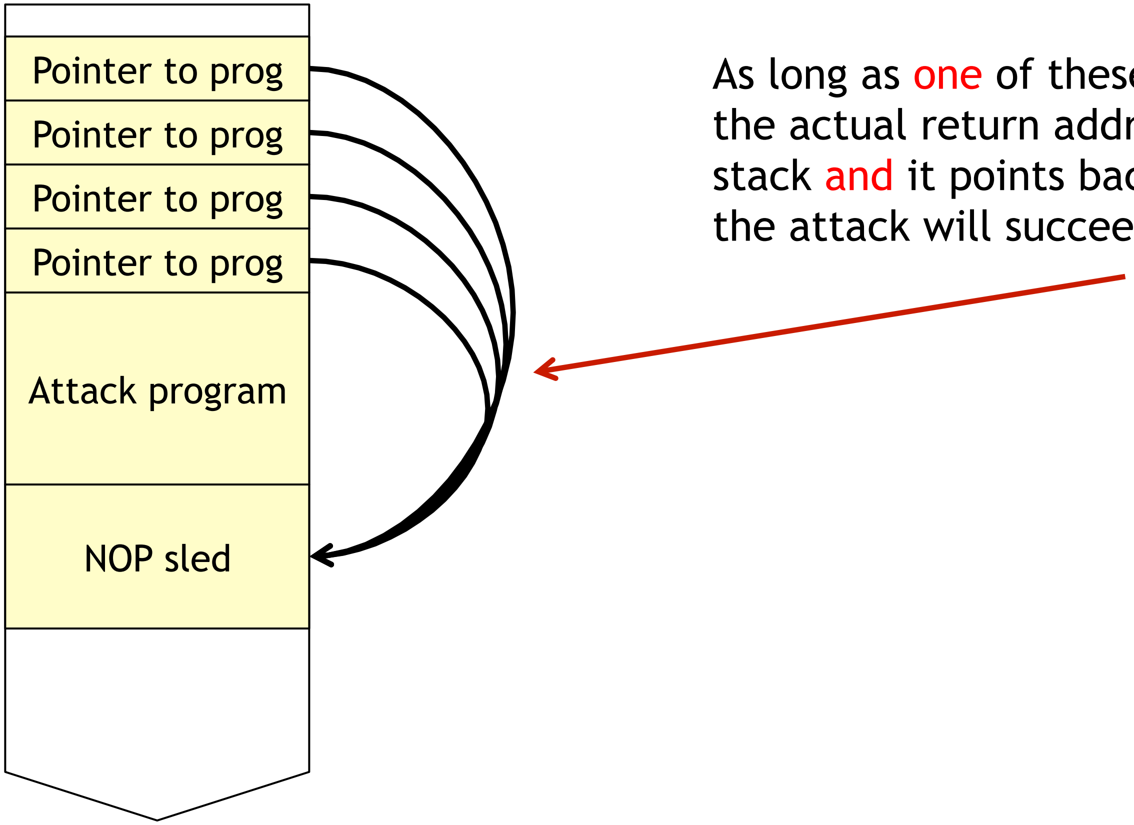
We can point anywhere in here



Unfortunately, we still need to know the exact location of the return address, which can be tricky…

We can also guess the location of the return address on the stack

How? Simply inject several new return addresses onto the stack!



As long as one of these pointers overwrites the actual return address location of the stack and it points back into the NOP sled, the attack will succeed

So… How do we defend against buffer overflows?

One way to stop buffer overflows is to use non-executable stacks

- If nothing can be executed from the stack, the exploit will fail
- Unfortunately, this is hard to implement in practice, as legacy compilers tend to like putting executable code in the stack
- A variation of this approach involves making writable memory pages non-executable

Another defense mechanism is the use of canary words

- A canary word is a random word of data written just before the return address on the stack
- Odds are, if the return address is overwritten, so is the canary word
- See Crispin Cowan's *StackGuard* for more information

So… How do we defend against buffer overflows?

Some researchers have proposed the use of address space randomization (ASR) to protect against stack smashing [1,2]

- Intuition: Shuffle the locations of variables and static data stored on the stack randomly each time the program is executed
- If the buffer is overflowed, there’s really no way* to tell what data structures will be overwritten
- Unfortunately, researchers have shown that this certain variations of this approach can be defeated [3]

Another way to stop stack smashing is to use a type-safe programming language (e.g., Java or C#) ☺

[1] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits,” 14th USENIX Security Symposium, August 2005.

[2] http://pax.grsecurity.net

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Bohen, “On the Effectiveness of Address-Space Randomization,” 11th ACM Conference on Computer and Communications Security, pages 298-307, Oct. 2004.

Stack smashing has a number of variants

If an attacker cannot launch a "traditional" stack smashing attack, there are other avenues that she could follow

Heap smashing is like stack smashing, but corrupts heap data

- There is much more variety across systems with how the heap is managed
- Attacks are difficult to construct, and typically are specific to a specific machine and/or configuration

Return to libc attacks jump to existing library code, not attacker code

- The attacker first correctly places her arguments on the stack
- Then she sets the return pointer to jump to pre-loaded libraries (e.g., libc)

Return oriented programming generalizes return to libc

- Generate a Turing complete library of computational "widgets" from the libc library (or any other library)
- Inject an attack vector that puts widgets together in some desirable way
- The effect is running arbitrary code without injecting executable code!

E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," CCS 2008.

Buffer Overflow Wrap-Up

In a stack smashing attack, the attacker alters kernel data structures to make a process behave in an unexpected manner

If the process that is attacked is running with elevated privileges (e.g., a server running as root), then the attacker can do all sorts of fun things

- Set up tunnels into the system
- Alter the password file
- Install rootkits to gain long term control of the system
- Delete files
- …

Clearly, this constitutes a violation of integrity of mechanism

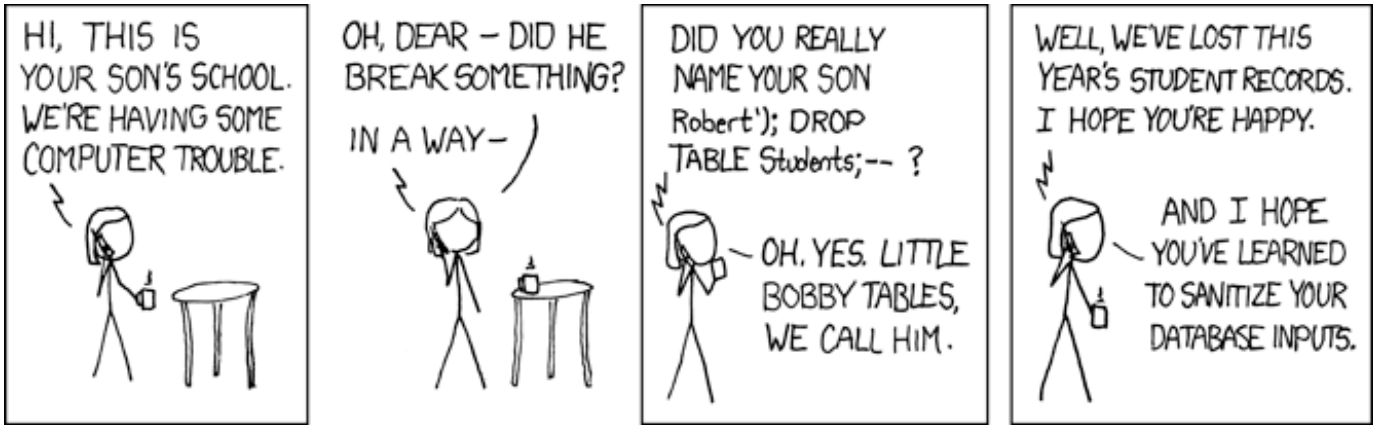
What about violating constraints within an application?

Sometimes, we don't need to subvert the entire OS to wreak havoc.

For example, many times applications have their own security mechanisms in place

- These constraints control the functionality exposed to different users
- Enforced by the application, not the OS
- In this case, compromising the application is all that is needed

Case Study: SQL Injection

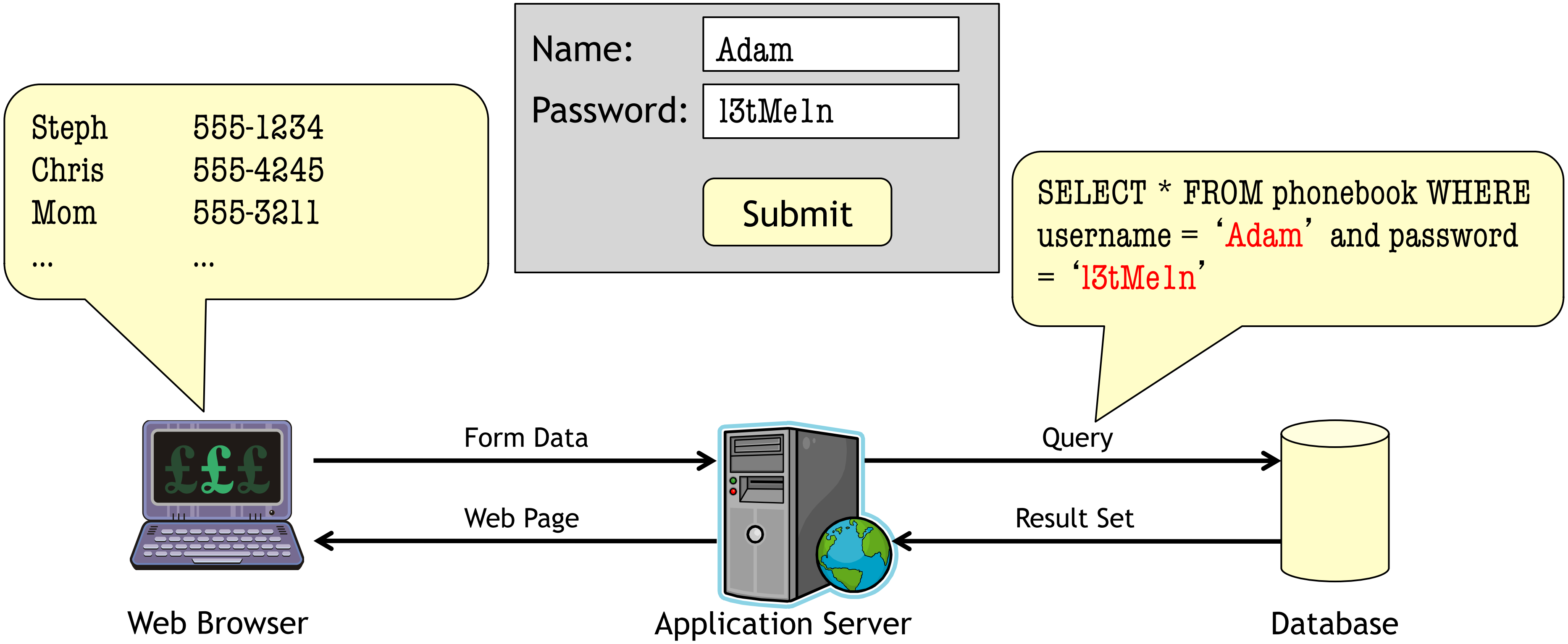

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

http://xkcd.com/327/

SQL injection attacks are specific to application servers with database back ends

These applications serve dynamic content based upon queries that are parameterized by user input

Example: An online phone book application



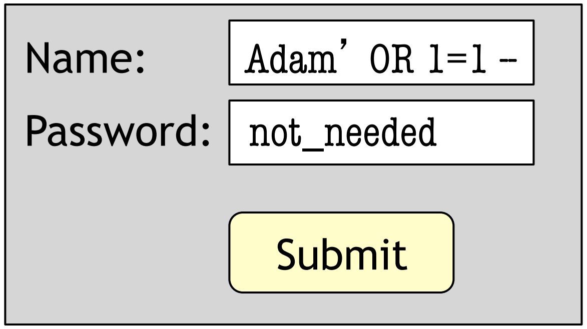
Steph	555-1234
Chris	555-4245
Mom	555-3211
…	…

Name: Adam

Password: l3tMeln

```
SELECT * FROM phonebook WHERE
username = 'Adam' and password
= 'l3tMeln'
```

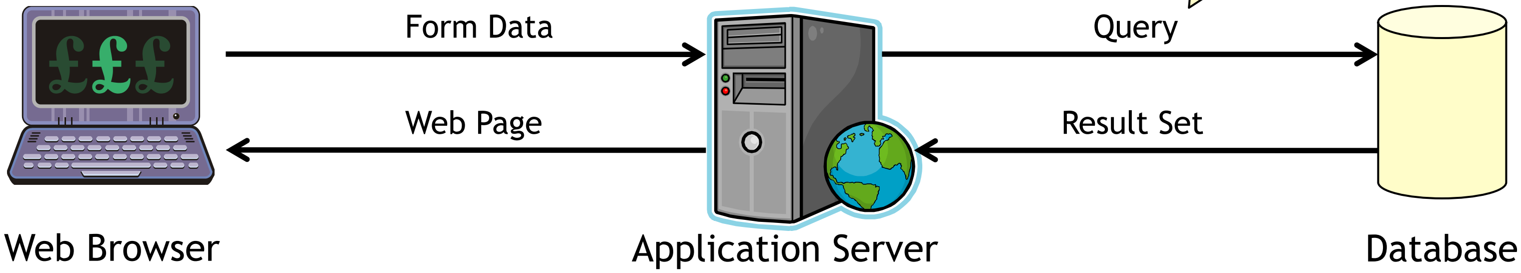
What could possibly go wrong with such a simple application?



Name: Adam’ OR 1=1 –

Password: not_needed

```
SELECT * FROM phonebook WHERE
username = ‘Adam’ OR 1=1 - -’
and password = ‘not_needed’
```



Note that:

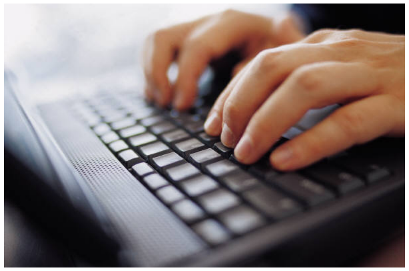
- The test 1=1 is always true!
- Also, “--” denotes a comment is SQL, so the password check is bypassed!

Result: All rows are displayed!

SQL injection can be used for all kinds of attacks…

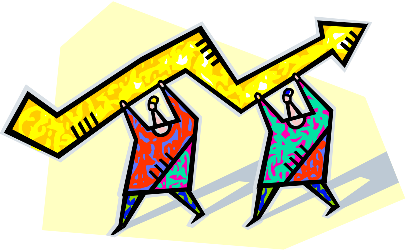
Gaining access to the system

- Perform an INSERT on the “Users” table



Modify data in the database

- Drop tables
- Change salaries
- Etc.



Perform privilege escalation attacks

- Stealing passwords
- Inserting rights
- Etc.

Clearly, this is much easier than crafting a buffer overflow!

The real difficulty lies in ascertaining the structure of the target database, but even this is not so difficult… So how we defend ourselves?

One solution is to sanitize database inputs

- For example, ‘ → \’, “ → \”, and \ → \\
- Unfortunately, this doesn’t always work…
  - SELECT * FROM salary WHERE id = 23 OR 1=1

Check input field validity

- Fields typically have fixed vocabularies
- Make sure illegal characters don’t appear
- Unfortunately, this isn’t always possible

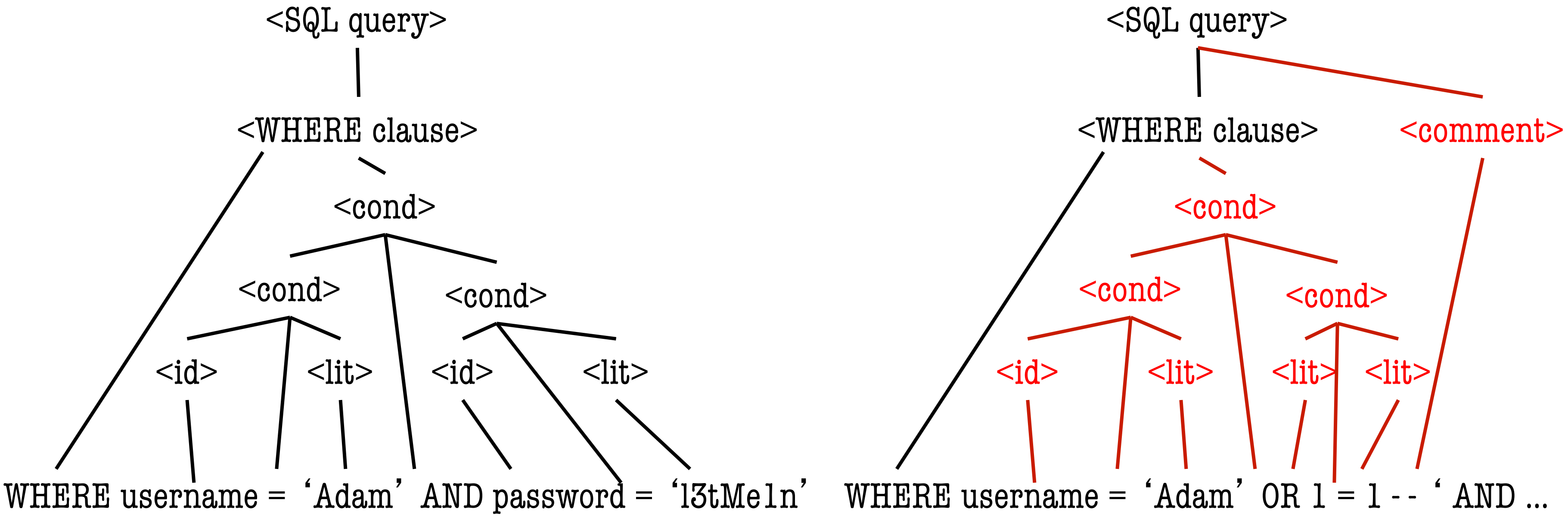
Place length limits on the allowed inputs to form fields

These are all basically hacks…

A solution proposed by Bandhakavi et al. uses parse tree comparison to detect SQL injection

Intuition: The parse tree for an SQL injection query is almost always different than the parse tree for the programmer intended query

Example:



Sruthi Bandhakavi, Prithvi Bisht, Madhusudan Parthasarathy, V.N. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," 14th ACM Conference on Computer and Communications Security (CCS), November 2007.

The problem that must be addressed is how to infer the programmer intended parse tree for a given query

Bandhakavi et al. solve this problem by generating benign candidate inputs for each dynamically generated query

The problem of how to generate inputs that are benign and do not alter the program's control flow is undecidable in general…

However, simple heuristics make this possible!

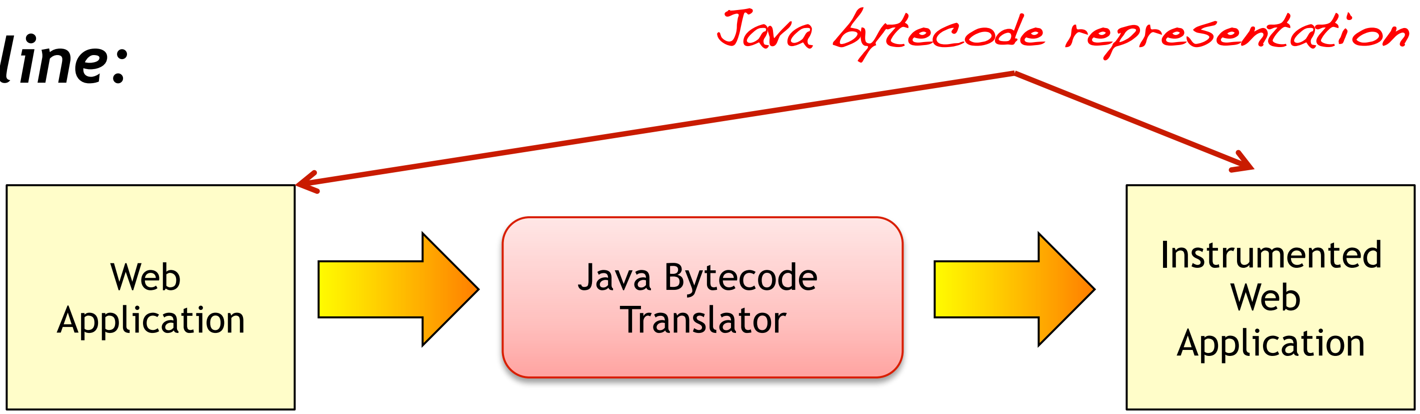
- Replace all integer inputs with "1"
- Replace all string inputs with an equal length string containing only "a"s
- Replace all booleans with "true"
- Make control flow decisions based upon actual input only

Parse trees are generated for the actual query and the candidate query

- Actual query is issued only if both parse trees match

How is the CANDID system set up?

Offline:



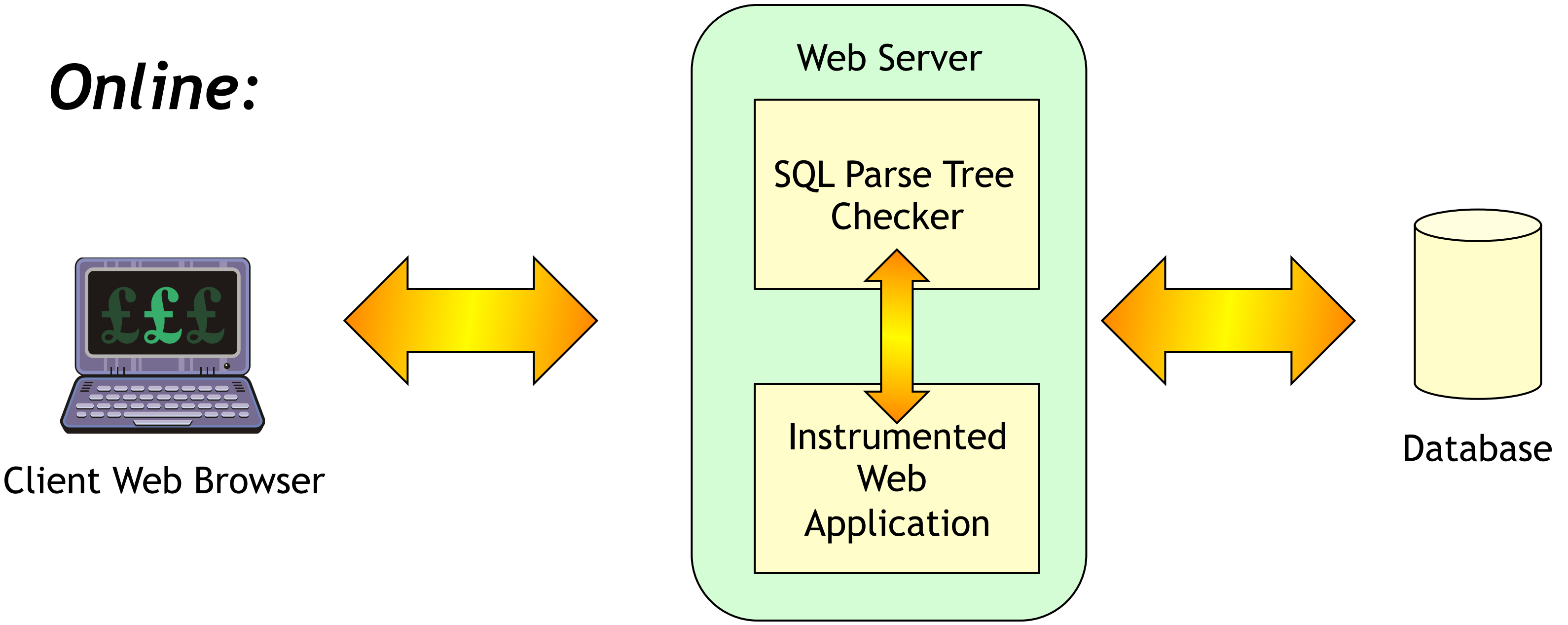
Java bytecode representation

Web Application

Java Bytecode Translator

Instrumented Web Application

Online:



Client Web Browser

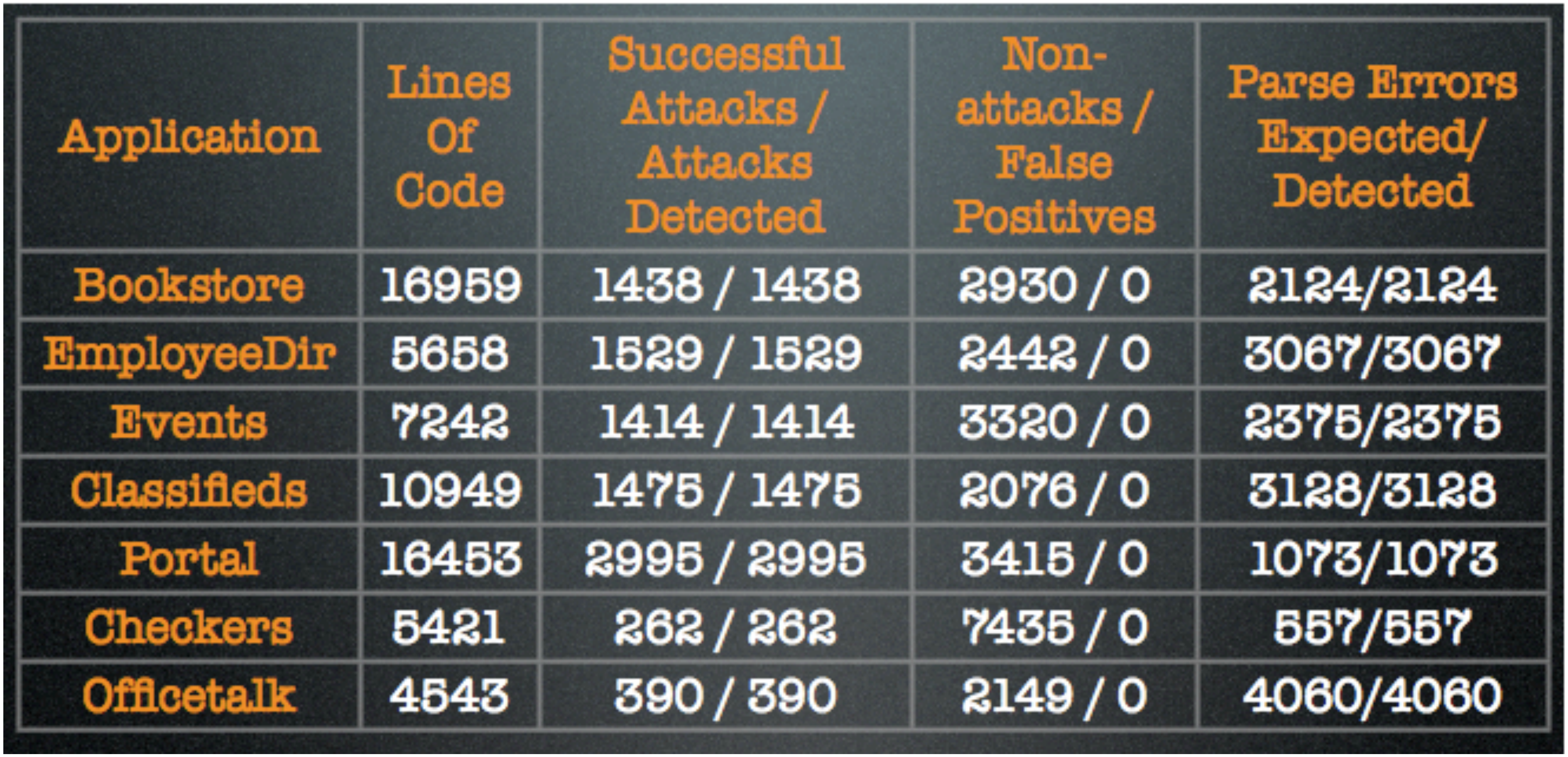
Web Server

SQL Parse Tree Checker

Instrumented Web Application

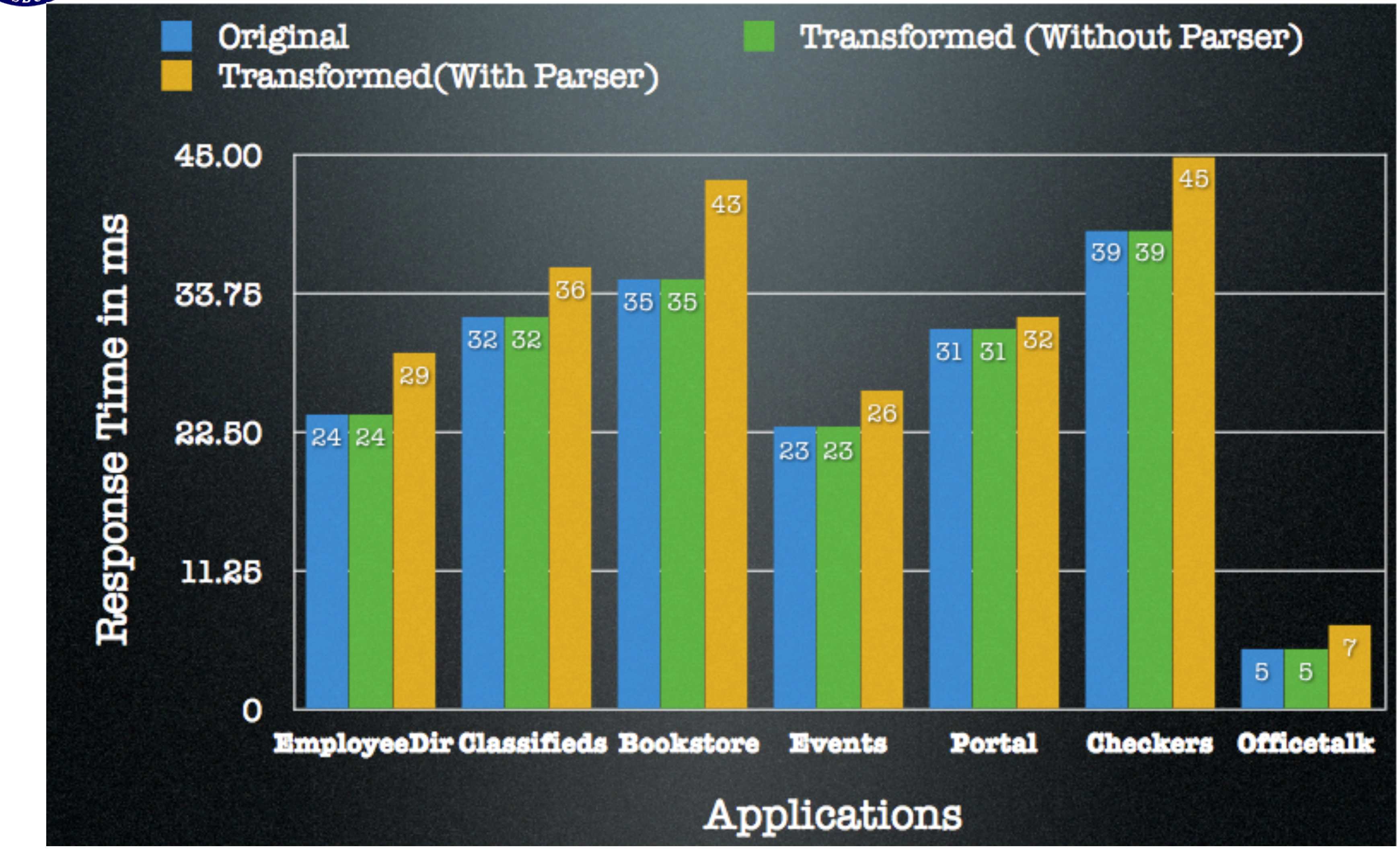
Database

Bandhakavi's solution performs well on standard database benchmarks



Application	Lines Of Code	Successful Attacks / Attacks Detected	Non-attacks / False Positives	Parse Errors Expected/ Detected
Bookstore	16959	1438 / 1438	2930 / 0	2124/2124
EmployeeDir	5658	1529 / 1529	2442 / 0	3067/3067
Events	7242	1414 / 1414	3320 / 0	2375/2375
Classifieds	10949	1475 / 1475	2076 / 0	3128/3128
Portal	16453	2995 / 2995	3415 / 0	1073/1073
Checkers	5421	262 / 262	7435 / 0	557/557
Officetalk	4543	390 / 390	2149 / 0	4060/4060

Bandhakavi's solution performs well on standard database benchmarks



Conclusions

OS Security relies on separating the trusted from the untrusted

The assumption of integrity of mechanism is central to security

Integrity of mechanism can be violated at the OS or application level

- OS example: Buffer overflow
- Application example: SQL injection