

18732: Secure Software Systems

Final Comments

(...and comments about the final)

Lujo Bauer Anupam Datta

CMU
Fall 2010

Four Broad Topics

0. Attacks

1. Software Security Architectures
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Learning Outcome

- Understanding of the state-of-the-art in control hijacking and web-based attacks and associated defenses

PROJECT 1

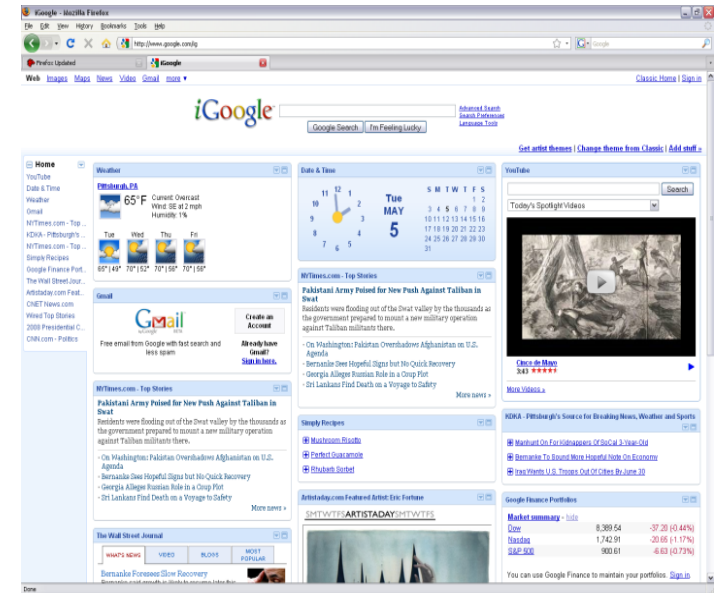
- *Note:* These two classes of attacks currently top the charts of common security vulnerabilities databases

Control Hijacking Attacks

- Attacker causes arbitrary attack code (e.g. root shell) to execute on target machine by hijacking the control flow of an application program
- Examples
 - Buffer overflow attacks
 - Integer overflow attacks
 - Format string vulnerabilities

Web Attacks

- Web-based attacks
(Top 3 web site vulnerabilities)
1. Cross-site scripting (XSS)
 2. Cross-site request forgery
 3. SQL injection



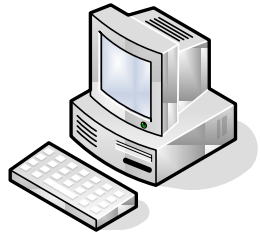
Four Broad Topics

0. Attacks
1. **Software Security Architectures**
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Learning Outcome

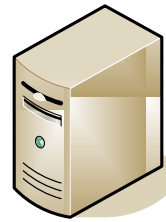
- Systematic thinking about system security
 - What is the security mechanism?
 - What is the adversary model?
 - What is the desired security property?
 - Does system guarantee security property in the presence of adversaries?
 - What is the trusted computing base (TCB)? What are our assumptions about the TCB?
- Understanding of specific security architectures for trusted computing and separation

Trusted Computing

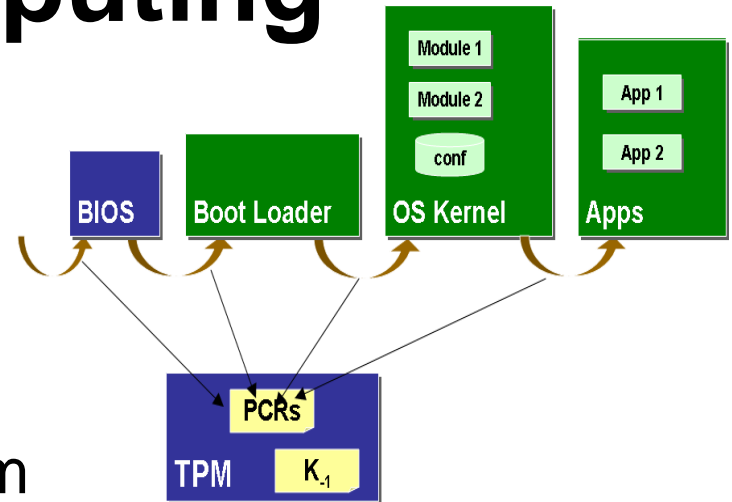


Verifier

What code are
you running? →



Remote platform

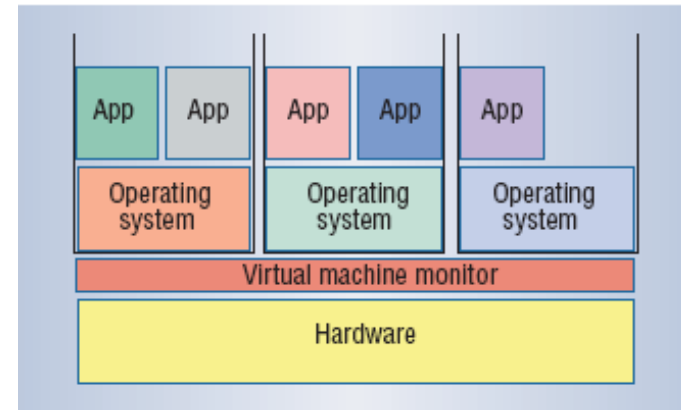


Take-home: Thinking like a security person!

- **Mechanism:** H/W-based protection + cryptographic protocol
- **Adversary:** local system + network
- **Semantic security property:** code integrity

Separation

- Virtual machine-style architecture aims to provide *isolation* between multiple VM's running on the same physical machine



- Other approaches and their implications for security
 - Hardware-based Memory Protection: segmentation, paging
 - Software-based Fault Isolation
 - Java Virtual Machine
 - Web Browser Security

JVM Model:

- **Mechanism:** stack inspection
- **Adversary:** “untrusted” principal
- **Semantic security property:** authorization policy

Four Broad Topics

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Security Analysis of Software

```

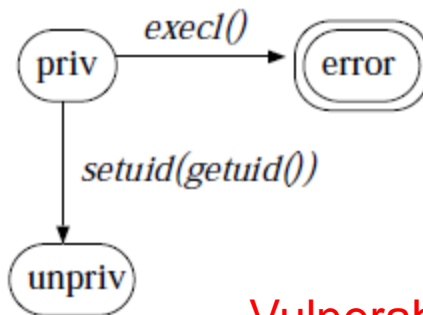
#include <stdio.h>
main(argc, argv)
int argc;          /* the number of arguments in argv */
char * argv[];    /* of null-terminated strings also known */
                  /* as an array of arrays of char. It */
                  /* holds the contents of the command line */
{
    int loopcounter;

    /* NOTE: The name with which the program */
    /* was invoked is held in argv[0]. */

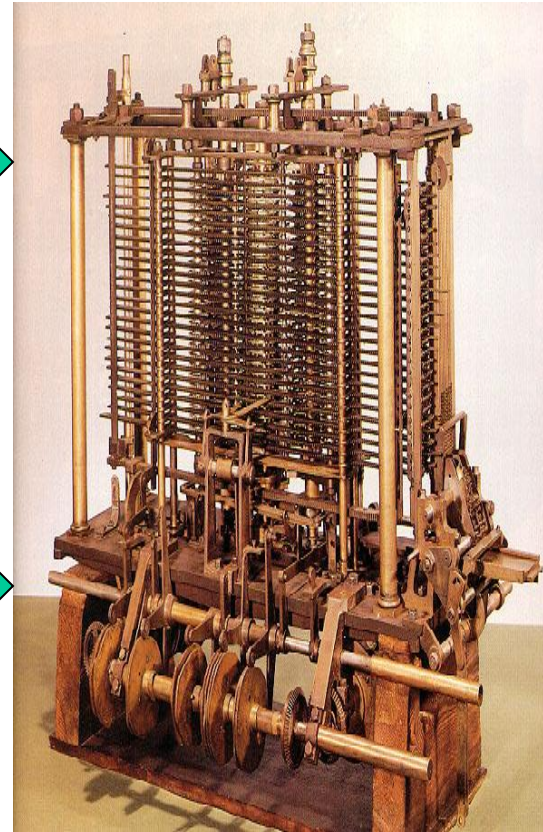
    for (loopcounter=1; loopcounter < argc; loopcounter++) {
        switch (argv[loopcounter][0]) {
            case '-': {
                printf("OPTION %s\n",
                    &argv[loopcounter][1]);
                break;
            }
            default : {
                printf ("FILENAME %s\n",
                    argv[loopcounter]);
                break;
            }
        }
    }
}

```

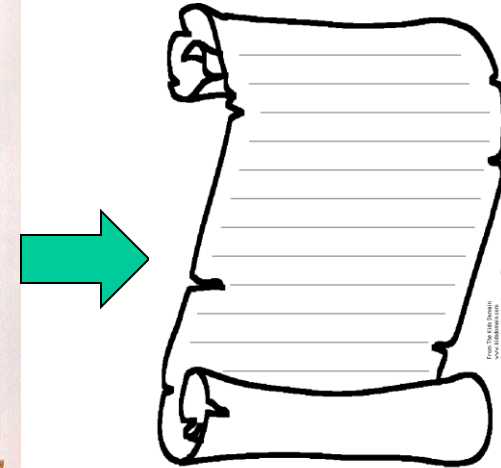
Program



Vulnerability
description



Analytical
Engine



List of potential
vulnerabilities
in program

Learning Outcome

- Understanding of *general methods* of software analysis (model checking, static analysis, dynamic analysis) and their application to identifying *security vulnerabilities* in legacy code written, for example, in C (buffer overflows, race conditions, setuid, chroot, malware...)
Note: Methods could be applied to tomorrow's attacks
- Understanding of strengths and limitations of various methods (false positives, false negatives, ...)
- Hands-on experience with a state-of-the-art industrial software analysis tool – *Coverity Prevent*

PROJECT 2

Security Analysis of Software

- Static analysis method for detecting buffer overflows
 - Targets one specific vulnerability
- Dynamic analysis
 - Is a program statement reachable? (Generality)
 - DART, EXE
- Software model checking for security
 - Is a program statement reachable? (Generality)
 - ASPIER: concurrent programs + adversary + semantic security properties (confidentiality, authentication)
 - MOPS: sequential programs + syntactic properties (sequence of function calls) + highly scalable

Security Analysis of Software

- Static analysis for bug-finding
 - Coverity Prevent: **syntactic properties** + scalable + usable
- Malware analysis
 - Reduction to a program analysis problem
 - **Semantic property**

Comparison: Metacompilation, Dynamic Analysis, Software Model Checking

	Meta-compilation	Dynamic analysis	SW Model checking
Automated	+++	++	+
Scalability	Millions of LoC	++	+
Richness of props	Shallow	++	+++
False positives	Lots	+++	++
False negatives	Lots	++	+++
Always terminates	Yes	No	No

Four Broad Topics

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Learning Outcome

- Understanding of basic concepts for type systems – syntax, static and dynamic semantics, type safety
- Understanding the security property of *non-interference* and how type soundness can imply non-interference
- Understanding of type systems to improve security of mobile (PCC) and assembly code (TAL)
- Hands-on experience with building a secure application in Jif – a security-typed programming language that extends Java with security types

Language-based Security

- Type-safe programming languages
 - Think Java (as opposed to C)
 - Better protection for language abstractions such as arrays
 - No buffer-overflow attacks on Java programs
- Security-typed programming languages
 - Language designed so that a program that compiles correctly (type checks) is “secure”

Information flow

- Non-interference (NI):
 - *General* semantic definition of security for secure systems
 - Program should permit *no* information flows from **HI** to **LO** security levels
 - Adversary controls **LO** variables
 - Too strong for many apps (hence declassification)
- Enforcement via type checking
 - **Theorem:** If program type checks, then it satisfies NI
 - Volpano-Smith-Irvine type system
 - Goal: Provide **language design** perspective

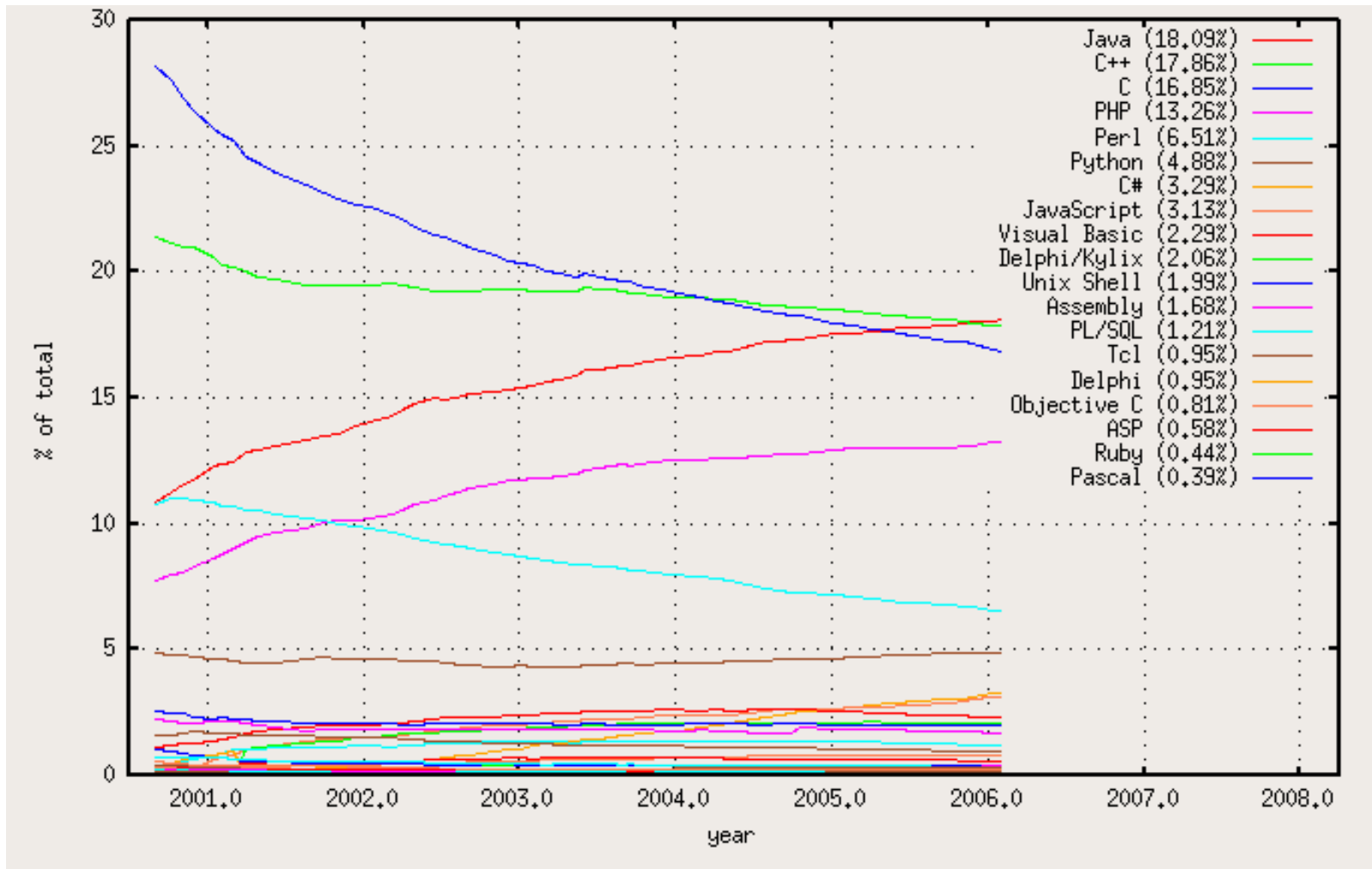
Information Flow: Jif

- Jif: Extending Java with information flow control
- Goal of lecture and assignment 3:
 - Provide **application building** perspective and experience in security typed languages

More Type Systems...

- Type systems for mobile and low-level code
 - Proof Carrying Code
 - Typed Assembly Language
 - Focus on type safety (\Rightarrow memory safety + control flow safety), not semantic security (e.g. non-interference, authorization)

Languages in Common Use



Four Broad Topics

0. Attacks
1. Software Security Architectures
2. Security Analysis of Software
3. Language-based Security
4. Run-time Security Enforcement

Learning Outcome

- Understanding of fundamentals of run-time enforcement of security properties
 - What class of security policies can be enforced using run-time monitoring?
 - How do we specify policies for run-time monitors to enforce?
- Understanding of a general run-time mechanism for guaranteeing control flow integrity (CFI)
 - CFI guarantees absence of control hijacking attacks

Run-time Security Enforcement

- Enforce security using mechanisms that monitor systems as they *execute* (as opposed to *static* analysis of code)
 - Captures properties that cannot be precisely enforced statically
 - E.g., properties that depend on specific inputs
- Some examples
 - *Stackguard*: Dynamic checks to prevent buffer overflows
 - Stack inspection, firewalls, applet sandboxing, displaying security warnings, OS reference monitors, virtual machines, software-based fault isolation, ...

Security Automata & CFI

- Security automata
 - Program executions produce traces (sequence of actions)
 - Take home: [Generality] Can enforce safety properties and beyond (e.g. infinite renewal)
- Control flow integrity
 - Provable method for ensuring that program execution respects CFG (i.e. prevents control hijacking attacks) in the presence of adversary that controls data memory
 - CFI inserts dynamic checks via machine code rewriting that are checked at run-time
 - Take home: A general safety property that if satisfied implies absence of a class of attacks (not a point solution)

Design Principles for Secure Systems

(illustrated in this course)

- Economy of mechanism
 - Small TCB: trusted computing (SRTM vs DRTM), proof checker in PCC, TAL (compiler not trusted)
- Complete mediation
 - Program monitors for run-time enforcement (in VMM, web browser, CFI, SFI)
- Least privilege
 - Web browsers (separation between rendering engine and kernel), NX data & NW code
- Provable security
 - Protocols, model checking, language-based security, run-time enforcement (soundness of analysis/enforcement); gap between model and actual system
- Usability
 - End-user security (e.g. using SSL), specification languages and analysis tools (e.g. composable policies, Prevent tool)

Final

- Wed, Dec 1, 10:30AM-12:00PM in class
- Scope:
 - Post-midterm lectures and readings only
(Oct 25 – Nov 29)

Thanks!

Good luck!