Language Based Security

Christian W. Probst
Informatics and Mathematical Modelling

The Problem with the Apple …

Overview

- Language Based Security
- Buffer Overflows
- Sandboxing
- Verification

The Poison… (ack. Dan Grossmann)

- Tossing together 20.000.000 lines of code
- From 1.000s of people at 100s of places
- And running 10.000.000s of computers holding data of value to someone
- And any 1 line could have arbitrary effect

Decontaminating the Apple

- Analyze the code
  - before execution [reject]
- Rewrite the code
  - before execution [modify]
- Monitor the code
  - during execution [stop before harm]

- In the past: isolated computing systems
  - updates done by an experienced administrator.
  - few programs.
  - physical access.
  - crashes and outages were just an nuisance.
- Today: the Internet
  - constant updates – sometimes applied automatically.
  - do you know the processes running on your machine?
  - a hacker in the Philippines is as close as your neighbor.
  - everything is executable.
  - everyday live depends on the infrastructure.

First Principle

Least Privilege

each principal is given the minimum access needed to accomplish its task.

[Saltzer & Schroeder '75]

- Language-based security can support an extensible protected interface
  - E.g., Java or MS CLR security

Second Principle

Keep the *TCB* small

Trusted Computing Base (TCB) :
components whose failure compromises the security of a system

- operating system:
  - kernel, memory protection system, disk image
- Small/simple TCB:

  ⇒ TCB correctness can be checked/tested/reasoned about more easily
  ⇒ more likely to work

Language Based Security

- Usually: security mechanisms treat programs as black box
  - Encryption
  - Firewalls
  - System calls
  - Access control
- security needs are not addressed
  - Downloaded, mobile code
  - Buffer overflows and other safety problems
  - System-level security validation

Language Based Techniques

Programs don't have to be black boxes!

1. Analyze programs at compile time or load time to ensure that they are secure
2. Check analyses at load time to reduce TCB
3. Transform programs at compile/load/run time so that they don't violate security, or so that they log actions for later auditing

Language Based Safety

- Restrict the language such that the programmer cannot do anything unsafe
  - No pointers
  - Bounds Checking
  - Type safety
  - Automated memory management
  - Access checking

Language Based Security

- Restrict the environment such that the program cannot do anything insecure
  - Sandbox
  - Signed code
  - Bytecode verifier
    - Checks for forged pointers, access violations, type safety violations
  - SecurityManager class – validates operations
  - ClassLoader – safe loading of classes

Language Based Security

- Restrict the environment such that the program cannot do anything insecure
  - Sandbox
  - Signed code
  - Bytecode verifier
    - Checks for forged pointers, access violations, type safety violations
  - SecurityManager class – validates operations
  - ClassLoader – safe loading of classes

Buffer Overflows

Buffer Overflows

```
void updateEnv(char *str)
{
  char *tmp;
  tmp = getenv("HOME");
  if (tmp != NULL)
    strcpy(str,tmp);
}
```


Buffer Overflows

```
void updateEnv(char *str, size_t size)
{
   char *tmp;
   tmp = getenv("HOME");
   if (tmp != NULL) {
      strncpy(str,tmp,size-1);
      str[size -1] = '\0';
   }
}
```

Buffer Overflows

- caused by untyped addresses
- a pointer is a pointer is a pointer
- no information about the size of the memory chunk pointed to

- strongly typed languages to the rescue
- a pointer is the address of an object or an array of objects
- array length is known (statically or dynamically)

1988: Morris Worm

Penetrated an estimated 6,000 machines (5-10% of hosts at that time)

Used a number of clever methods

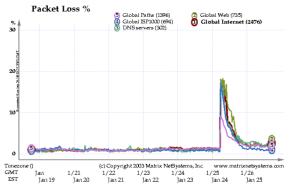
- brute force password guessing
- bug in default sendmail configuration
- X windows vulnerabilities, rlogin, etc.
- buffer overrun in fingerd

Remarks:

- Buffer overruns account for roughly half of CERT advisories.
- You'd think we could solve this but…

2003: MS-SQL Slammer worm

- Jan. 25, 2003: SQL and MSDE servers on Internet turned into worm broadcasters
  - Buffer-overrun vulnerability
  - Many others (e.g., Code Red)
  - But this was the worst yet: spread across entire net in 10 min!
- Can't rely on users to patch!
  - Patch had been out for 6 months
  - Infected SQL servers at Microsoft itself
  - May only have 10 min to develop next patch…
- We need automatic prevention.
  - Lots of work on catching buffer overruns at compile time using various kinds of type systems, analyses, tools, etc.



1999: Love Bug & Melissa

Both email-based viruses that exploited:

- a common mail client (MS Outlook)
- trusting (i.e., uneducated) users
- VB scripting extensions within messages to:
  - lookup addresses in the contacts database
  - send a copy of the message to those contacts
  - ran with full privileges of the user

Melissa: hit an estimated 1.2 million machines.

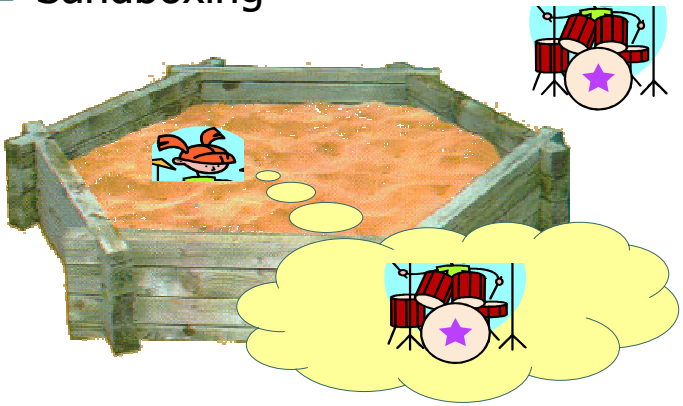
Love Bug: caused estimated $10B in damage.

Sandboxing

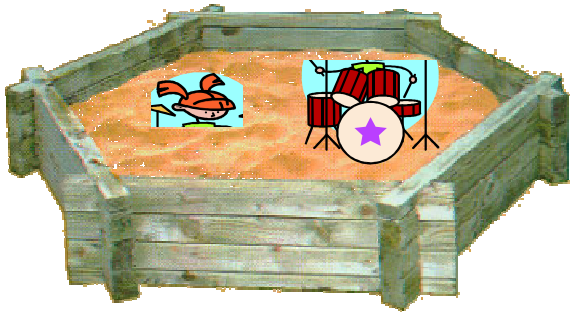
Sandboxing

Sandboxing

Sandboxing



Sandboxing



Sandboxing

Sandboxing

Sandboxing

Sandboxing

Sandboxing

Sandboxing

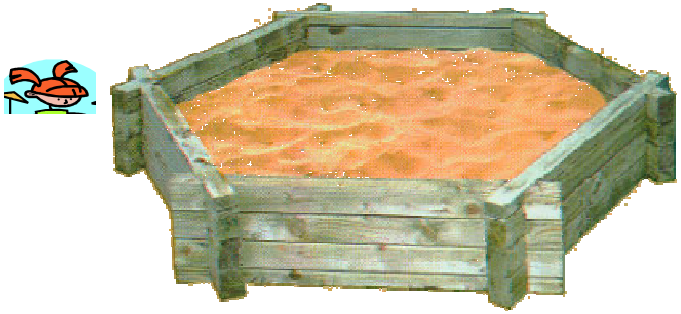
- implemented in the Java Virtual Machine (among others)
- each applet gets its own play ground
- VM manages
  - loading of applets
  - granting/revoking of resources

Controllable Properties

- theoretically unlimited
  - memory access
  - resources
  - network communication
  - file access
- dynamic properties

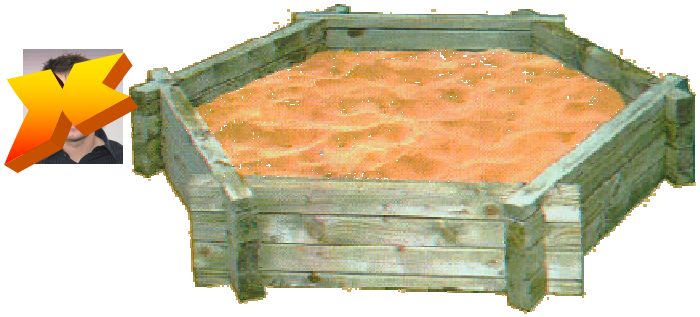
Verification

Verification



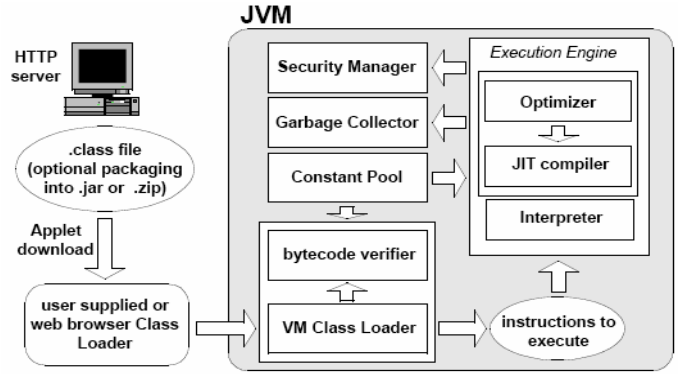
Verification

Verification



Verification

- pioneered by the Java Virtual Machine
- Problem: running arbitrary code downloaded from the network is dangerous
- Idea: let's not allow the programmer to do insecure things (duh)
  - Safety
  - Security
- check certain properties of incoming programs
  - e.g. stack will not grow unbounded



Verifier Essentials

- checks a piece of code for type consistency and other properties
- least conditions for bytecode to be accepted
  - Type correctness
  - No stack overflow or underflow
  - Code containment (no wild jumps)
  - Local variable initialization
  - Object initialization

Verifier Essentials

- properties can be checked dynamically at runtime
  - high performance penalty
- verifier checks properties statically before execution
- verifier is actually a data flow analyzer
- contains an abstract interpreter that works on ***types*** instead of ***values***

The Verifier Algorithm

```
1:  todo ← true
2:  while todo = true do
3:     todo ← false
4:     for all i in all instructions of a method do
5:        if i was changed then
6:           todo ← true
7:           check whether stack and local variable types match definition of i
8:           calculate new state after i
9:           for all s in all successor instructions of i do
10:             if current state for s ≠ new state derived from i then
11:                assume state after i as new entry state for s
12:                mark s as changed
13:             end if
14:          end for
15:       end if
16:    end for
17: end while
```



```
.class B
.method public to_int(LA;)I
 .limit stack 3
 .limit locals 3

   aload_1
   ireturn

.end method
```

Registers	Stack
R0 this	empty
R1 A	
R2 ?	


```
.class B
.method public to_int(LA;)I
 .limit stack 3
 .limit locals 3

   aload_1
   ireturn

.end method
```

Registers	Stack
R0 B	empty
R1 A	
R2 ?	


```
.class B
.method public to_int(LA;)I
 .limit stack 3
 .limit locals 3

   aload_1
   ireturn

.end method
```

Registers	Stack
R0 B	A
R1 A	
R2 ?	

```
.class B
.method public to_int(LA;)I
 .limit stack 3
 .limit locals 3

   aload_1
   ireturn

.end method
```

Verifier error - expected to find an inte
on the stack

Verifiable Properties

- all statically testable properties
  - typing
    - variables
    - stack cells
  - stack size
  - initialization
- must hold for ALL possible executions of the program

Conclusion

- Language-based Technologies allow to restrict vulnerabilities by
  - restricting programming language constructs
    - strongly typed languages
  - restricting execution environments
    - sandboxing
    - verification
- Implement two important principles
  - least privilege
  - small TCB