Plan for Today

- Recap: Java Platform Security
- Trusted Computing Base: should we trust Java’s?
- Hair-Dryer Attacks

Project Design Documents

1. A description of your project: what it will do and why it is useful, fun, or interesting.
2. A high-level description of your design, including a module dependency diagram showing the most important modules.
3. A description of your implementation and testing strategy including:
   - how you will divide the work amongst your team
   - how you will order the work to support incremental development
   - how you will do unit testing and integration testing
   - a list of milestones and a schedule for achieving them, leading to a completed project on December 7
4. A list of questions

Due: on paper, beginning of class Tuesday

Schedule Design Review meetings [link on course site]

Recap: Java Platform

Running Mistyped Code

> java Simple
Exception in thread "main" java.lang.VerifyError:
  (class: Simple, method: main signature:
   ([Ljava/lang/String;)V)

Register 0 contains wrong type

Running Mistyped Code

> java –noverify Simple
Exception in thread "Simple" java.lang.VerifyError:
  (class: Simple, method: main signature:
   ([Ljava/lang/String;)V)

Register 0 contains wrong type

Running Mistyped Code

> java –noverify Simple
Exception in thread "Simple" java.lang.VerifyError:
  (class: Simple, method: main signature:
   ([Ljava/lang/String;)V)

Register 0 contains wrong type

Running Mistyped Code

> java –noverify Simple
Exception in thread "Simple" java.lang.VerifyError:
  (class: Simple, method: main signature:
   ([Ljava/lang/String;)V)

Register 0 contains wrong type

Running Mistyped Code

> java –noverify Simple
Exception in thread "Simple" java.lang.VerifyError:
  (class: Simple, method: main signature:
   ([Ljava/lang/String;)V)

Register 0 contains wrong type
Recap: Trusted Computing Base

- The part of the system that must be trusted to behave correctly for the desired security properties to be guaranteed
- Should we trust the Java platform TCB?

Building Trust

- Simplicity
  
  There are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. — Tony Hoare

- Extensive validation

  Boeing 787 Dreamliner delay conspiracy theories
  “Rather, he thinks avionics software is hung up by the effects of the RTCA/DO-178b standard, which certifies avionics software and in his opinion causes unnecessary delays in the delivery of same. In yesterday’s call, Boeing executives … downplayed the avionics software lag, but conceded they welcome more time to test it.”

- Design Process

  (Slide from Nate Paul’s ACSAC talk)

Verifier (should be) Conservative

Verifier (should be) Conservative

The Worst JVML Instruction

- jsr [branchbyte1] [branchbyte2]
- Operand Stack
  
  \[ \ldots \Rightarrow \ldots, \text{address} \]

- Description
  
  The address of the opcode of the instruction immediately following this jsr instruction is pushed onto the operand stack as a value of type returnAddress. The unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is \((\text{branchbyte1} \ll 8) \mid \text{branchbyte2}\). Execution proceeds at that offset from the address of this jsr instruction. The target address must be that of an opcode of an instruction within the method that contains this jsr instruction.

- Notes
  
  Note that jsr pushes the address onto the operand stack and ret gets it out of a local variable. This asymmetry is intentional.

(Slide from Nate Paul’s ACSAC talk)
Try-Catch-Finally

```java
public class JSR {
    static public void main(String args[]) {
        try {
            System.out.println("hello");
        } catch (Exception e) {
            System.out.println("There was an exception!");
        } finally {
            System.out.println("I am finally here! ");
        }
    }
}
```

Vulnerabilities in JavaVM

- **Years Since First Release**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Vulnerabilities Reported**: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45

![Graph showing vulnerability trend over years](image)

Where are They?

- **Verification**: 12
- **API bugs**: 10
- **Class loading**: 8
- **Other or unknown**: 2
- **Missing policy checks**: 3
- **Configuration**: 4
- **DoS attacks (crash, consumption)**: 5

Several of these were because of jsr complexity.

Low-level vs. Policy Security

**Low-level Code Safety**
- Type safety, memory safety, control flow safety
- Enforced by Java bytecode verifier and run-time checks in VM
- Needed to prevent malcode from circumventing any policy mechanism

**Policy Security**
- Control access and use of resources (files, network, display, etc.)
- Enforced by Java class
- Hard part is deciding on a good policy

Is this really the whole TCB?
**Bytecode Verifier**

Checks JVML code satisfies safety properties:
- Simulates program execution to know types are correct, but doesn’t need to examine any instruction more than once
- After code is verified, it is trusted: is not checked for type safety at run time (except for casts, array stores)

**Key assumption:** when a value is written to a memory location, the value in that memory location is the same value when it is read.

---

**Can you really blame cosmic rays when your program crashes?**

- IBM estimate: one cosmic-ray bit error per 256 megabytes per month
- For people running big datacenters, this is a real problem
- If your processor is in an airplane or in space risk is much higher

---

**Improving the Odds**

- Set up memory so that a single bit error is likely to be exploitable
- Mistreat the hardware memory to increase the odds that bits will flip

---

**Making Bit Flips Useful**

Fill up memory with Filler objects, and one Pointee object:

```java
class Filler {
    Pointee a1; Pointee a2; Pointee a3; Pointee a4; Pointee a5; Pointee a6; Pointee a7;
}

class Pointee {
    Pointee a1; Pointee a2; Filler f; int b;
    Pointee a5; Pointee a6; Pointee a7;
}
```

---

**Filling Up Memory**

Pointee p = new Pointee();
ArrayList<Filler> fillers = new ArrayList<Filler>();
try {
    while (true) {
        Filler f = new Filler();
        f.a1 = p; f.a2 = p; f.a3 = p; ...; f.a7 = p;
        fillers.add(f);
    }
} catch (OutOfMemoryException e) { ; }

---

**Violating the Assumption**

```java
... // The object on top of the stack is a SimObject
astore_0
// There is a SimObject in location 0
aload_0
// The value on top of the stack is a SimObject

If a cosmic ray hits the right bit of memory, between the astore andaload, the assumption might be wrong.
```
Wait for a bit flip...

- Remember: there are lots of Filler objects (fill up all of memory)
- When a bit flips, good chance (~70%) it will be in a field of a Filler object and it will now point to a Filler object instead of a Pointee object

**Type Violation**

After the bit flip, the value of f.a2 is a Filler object, but f.a2 was declared as a Pointee object!

Can an attacker exploit this?

Finding the Bit Flip

```java
Pointee p = new Pointee();
ArrayList<Filler> fillers = new ArrayList<Filler>();
try {
    while (true) {
        Filler f = new Filler();
        f.a1 = p; f.a2 = p; f.a3 = p; ... f.a7 = p;
        fillers.add(f);
    }
} catch (OutOfMemoryException e) { ; }
while (true) {
    for (Filler f : fillers) {
        if (f.a1 != p) { // bit flipped!

            // Do whatever you want! No security policy now...
            new File("C:\thesis.doc").delete();
        } else if (f.a2 != p) {
        ...
    }
}
```

Violating Type Safety

```java
Filler f = (Filler) e.nextElement();
if (f.a1 != p) { // bit flipped!
    Object r = f.a1; // Cast is checked at run-time
    Filler fr = (Filler) r; // Cast is checked at run-time
    fr.a1 = 1524383; // Address of the SecurityManager
    fr.a4.a1 = null; // Set it to a null
    // Do whatever you want! No security policy now...
    new File("C:\thesis.doc").delete();
}
```
Getting a Bit Flip

• Wait for a Cosmic Ray
  – You have to be really, really patient... (or move machine out of Earth’s atmosphere)
• X-Rays
  – Expensive, not enough power to generate bit-flip
• High energy protons and neutrons
  – Work great - but, you need a particle accelerator
• Hmm....

Using Heat

50-watt spotlight bulb
Between 80°-100°C, memory starts to have a few failures
Attack applet is successful (at least half the time)!
Hairdryer works too, but it fries too many bits at once

Should Anyone be Worried?

Verifier assumes the value you write is the same value when you read it
By flipping bits, we can violate this assumption
By violating this assumption, we can violate type safety: get two references to the same storage that have inconsistent types
By violating type safety, we can get around all other security measures

Recap

Verifier assumes the value you write is the same value when you read it
By flipping bits, we can violate this assumption
By violating this assumption, we can violate type safety: get two references to the same storage that have inconsistent types
By violating type safety, we can get around all other security measures