Defibrillating Web Security

we need better security tech!

Meder Kydyraliev, HackInTheBox 2012

Current State

Technology

- Strings are used to represent everything
- Most of the security related functionality is manual:
  - escaping
  - XSRF
  - AuthZ
- Some things are automated but still suck

HTML Escaping

- Typical advice: *escape user controlled input*
- Can be done manually or automatically

Automated Escaping

- Most common approach:
  - HTML escape everything!
- Lots of subtle problems with JavaScript

Subtle Problem #1

```
<script>
  // var foo = '<%= lname %>'
  var foo = 'O'Connell'
</script>
```

How do you escape?

```
var foo = 'O\'Connell'
var foo = 'O&#39;Connell'
```

Subtle Problem #2

Typical advice: *Avoid using* `innerHTML`!

```
d = document.getElementById('mydiv')

d.innerText = name; // O&#39;Connell
d.innerHTML = name; // O'Connel
```

Subtle Problem #3

```
<form ...>
  <input type="text" id="name"...
</form>
```

```
t = document.getElementById('name')
t.value = foo; // O&#39;Connell
```

To unescape entities…don't ask!

Escaping correctly

- Has to be contextual
- Has been implemented:
  - In Java, Mike Samuel's escaper: https://github.com/mikesamuel/html-contextual-autoescaper-java
  - Integrated with Rails by Ilya Grigorik: https://github.com/igrigorik/contextual

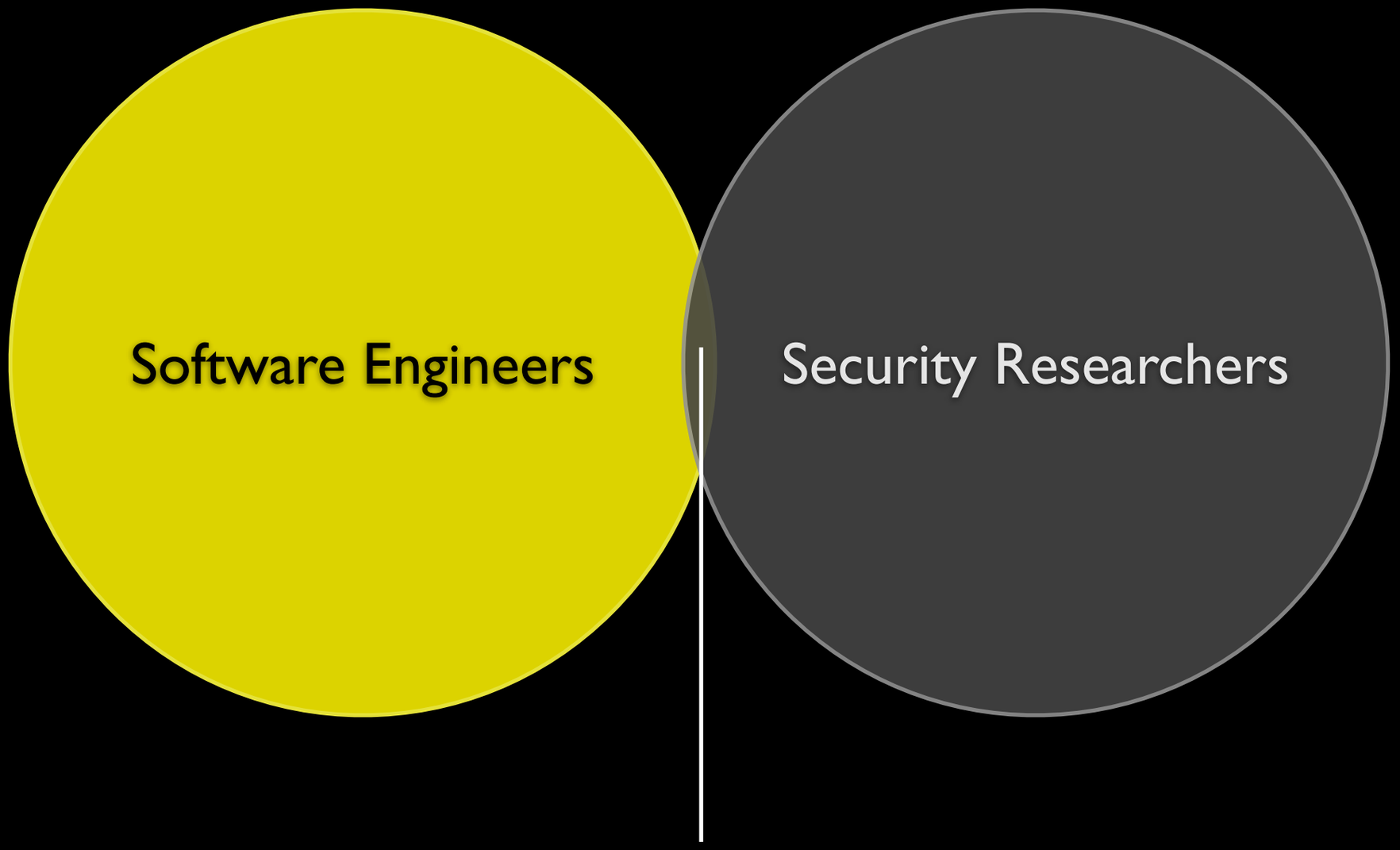
Other Awkward Solutions

- Authorization
- XSRF token handling
- Assumptions verifications

Security Industry

How many of you gave recommendations on addressing ***application*** security vulnerabilities in the past 6 month?

How many of you have written an application that was used by more than 100 people simultaneously?



This intersection(security engineering) needs to grow!

Current Industry

- Finding bugs brings more $$$ then solving classes of problems
- Complex software solutions built to address symptoms:
  - WAFs
  - Static analysis tools

Current Industry

- Solutions solving classes of bugs are evaluated as:
  - “Will this work on the app our interns wrote 5 years ago?”
  - “Will this work on our outsourced app?”
- A lot of recommendations, best practices and advice based on the 90s tech/mentality:
  - check all your integer operations for overflows
  - each time you copy something check length
  - etc

Strings

- Strings are everywhere!
- Strings! Strings! Strings!
- Another C artifact?
- Real languages have types!

~~Strings~~ Types

- Type hierarchy
- Ability to associate metadata
- Problems
  - habits
  - existing APIs that expect strings

Can we add “types” to Strings?

Kind of, remember tainting?

Taint Tracking Basics

Taint Tracking Basics

```
>> foo = "Got kumys?".taint
=> "Got kumys?"
>> foo.tainted?
=> true
>> $SAFE=1
=> 1
>> eval foo
SecurityError: Insecure operation -
   from (irb):13:in `eval'
       from (irb):13
>> f = File.new("/tmp/#{foo}")
SecurityError: Insecure operation - initialize
   from (irb):15:in `initialize'
```

Taint Tracking Basics

- Taint *source*: source of untrusted data (e.g. HTTP parameters)
- Taint *sink*: security sensitive function/ method (e.g. eval, file operations)
- Taint propagation:
  - `foo = “clean” + tainted`
  - `“clean with #{tainted}”`
  - `newtainted = tainted.gsub(...)`

Taint Tracking

- Perl has it, Ruby has it...nobody's using either
- Why?
  - inflexibility
  - binariness

Inflexibility

- Taint tracking systems usually part of a larger system...
- ...which tries to tackle problems that vast majority of applications do not have (e.g. untrusted code)
- Hardcoded rules that aren't always applicable or easily configurable

Binariness

- Strings are either tainted or not tainted
- What about:

```
locale = params[:locale]
...
help = read_file(“#{locale}/help.erb”)
...
Language: <%= locale %>
<%= help %>
```

How do we fix taint tracking?

Fixing taint tracking

- Make it practical
- Make it configurable
- Make it contextual

Practical

- Keep in mind that vast majority of apps
  - don’t run untrusted code
  - are client-server apps
- Try to solve problems that applications have

Configurable

- Let me choose
  - taint sources (e.g. I don't care about environment vars)
    - taint sinks (e.g. I may not care about File APIs at first)
- Let me configure untainting

Introducing Gravizapa!

Gravizapa

- Runtime contextual taint tracking system
- Prototypes implemented in Java and Ruby

Gravizapa Features

- Contextual
  - tainted strings are only untainted(marked safe) for particular context (e.g file path)
- Configurable
  - sources, sinks and cleaners specified in a config file
- No application changes required!

Java version

- Uses Java's ClassFileTransformer
  - Introduced in Java 5
  - Allows instrumentation of any class
    - no classloaders mess
    - can even modify JDK classes!
- Implemented by Josh Deprez (joshdeprez.com), intern @ Google

i13n of String, etc

- Java Strings are immutable (unlike Ruby)
  - Before people would patch rt.jar
    - and their JVM would crash
  - Java 5 agents to rescue

Java 5 Agent

- java -javaagent:...agent.jar
- Detailed docs:
  - http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/package-summary.html
- Agent will be called with bytecode of *new classes* begin loaded as well as classes *already loaded*!
- Allows modification of JRE classes
  - but not the schema (i.e. can't add new members)
  - but can modify any method code
- Used OW2 ASM bytecode instrumentation library

Tracking Taint

- Strings are immutable
- Instrumentation doesn't let you add new fields
- Where do we store taint data?

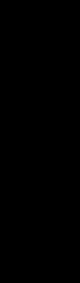
String

```
public final class String ...
{
    private final char value[];
    private final int offset;
    private final int count;
    ...
```

String

char value[]

Kumys, the best drink ever!



offset = 0

count = 27

String

char value[]

Kumys, the best drink ever!

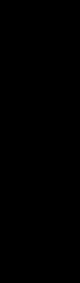
offset = 7

count = 20

String

char value[]

Kumys, the best drink ever!



offset = 0

count = 27

String's Taint Data

```
char value[]
```

TAINT **Kumys, the best drink ever!**

```
offset = 2

count = 29
```

String's Taint Data

- Contains
  - taint marker
  - contextual safety bits (e.g. SQL escaped, HTML escaped, etc)

Sources and Sinks

- Sources

```
public String getParameter(String name) {
  String paramValue = ...
  return Taint.markAsTainted(paramValue);
}
```

- Sinks

```
public File(String path) {
  Taint.checkTaint(path, FILE_PATH);
  ...
}
```

Taint cleaner

- Taint cleaners

```
public String htmlEscape(String str) {
  StringBuilder ret = new StringBuilder();
  for (int i; i < str.length(); i++) {
    ret.append(....);
  }
  return Taint.setSafeFor(HTML,
                          ret.toString());
}
```

Taint Propagation

- Mostly straightforward instrumentation
  - e.g. `toLowerCase()`, just add a call to mark return value as tainted
- “Kumys, ” + “Is” + “ The” + “ Best”:

```
new StringBuilder(“Kumys, ”)
    .append(“Is”).append(“ The”)
    .append(“ Best”).toString();
```

Configuration

```
   "my/http/HttpServletRequestImpl":{
    "getParameter(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SOURCE"
    }
},
"org/example/util/Sanitiser":{
    "sanitizePath(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SET_SAFETY",
      "safetyTags":["FilePath"]
    }
},
"java/io/FileReader":{
    "<init>(Ljava/lang/String;)V":{
      "modType":"TAINT_SINK_THROW",
      "methodParameters":[1],
      "safetyTags":["FilePath"]
    }
},
```

Configuration

```
    "my/http/HttpServletRequestImpl":{
    "getParameter(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SOURCE"
    }
  },
  "org/example/util/Sanitiser":{
    "sanitizePath(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SET_SAFETY",
      "safetyTags":["FilePath"]
    }
  },
  "java/io/FileReader":{
    "<init>(Ljava/lang/String;)V":{
      "modType":"TAINT_SINK_THROW",
      "methodParameters":[1],
      "safetyTags":["FilePath"]
    }
  },
```

Configuration

```
  "my/http/HttpServletRequestImpl":{
    "getParameter(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SOURCE"
    }
  },
  "org/example/util/Sanitiser":{
    "sanitizePath(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SET_SAFETY",
      "safetyTags":["FilePath"]
    }
  },
  "java/io/FileReader":{
    "<init>(Ljava/lang/String;)V":{
      "modType":"TAINT_SINK_THROW",
      "methodParameters":[1],
      "safetyTags":["FilePath"]
    }
  },
```

Configuration

```
   "my/http/HttpServletRequestImpl":{
    "getParameter(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SOURCE"
    }
},
"org/example/util/Sanitiser":{
    "sanitizePath(Ljava/lang/String;)Ljava/lang/String;":{
      "modType":"TAINT_SET_SAFETY",
      "safetyTags":["FilePath"]
    }
},
"java/io/FileReader":{
    "<init>(Ljava/lang/String;)V":{
      "modType":"TAINT_SINK_THROW",
      "methodParameters":[1],
      "safetyTags":["FilePath"]
    }
},
```

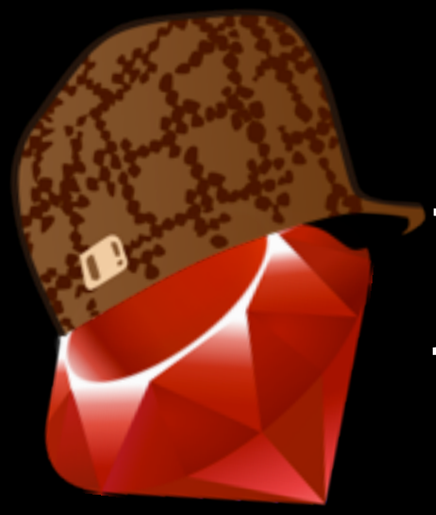
Bytecode i13n

- Bytecode with ASM is easier than Java code
  - `ARETURN`
- **Extremely** powerful facility, can be used to
  - implement authorization checks (ACLs, XSRF, etc)
  - assert style checks (e.g. has input been escaped?)
  - sandboxing

Ruby Gravizapa

Ruby

- Strings are mutable
  - `gsub!` vs `gsub`
  - Taint context propagation rules become a bit more complex
  - Monkey patch String and source/sink



Monkey patching

- Ruby promises that you can do anything, which is a lie!
- You CANNOT
  - monkey patch `gsub!` because it breaks capturing groups (e.g. `$1` won't work)
  - monkey patch string interpolation, e.g. "`My name is #{name}`"

Ruby String Interpolation

- Patched JRuby to invoke `pre_append()` if one exists in RubyString:

```
+ if (interpolation && this.respondsTo("pre_append")) {
+   otherStr = (RubyString) this.callMethod("pre_append", otherStr);
+ }
```

Ruby Gravizapa

- Code primarily aimed at demonstrating the concept
- May try to pitch the idea to Ruby 2

Java Gravizapa

- Needs more testing
- Will eventually be open-sourced
- More performance testing & optimizations

?

!