Pre-lecture Discussion
- Cell Phones
  - Mostly informal papers on vulnerabilities
  - Deal mostly with wifi
  - Mobile botnets not popular
    - Can’t Monetize?
    - Non-uniformity platforms, no easy single attack

Project
- Previously seen ideas from past classes
  - Blacklist/regexps – 0% successful
    - $s/<\?SCRIPT[^>]*>//g$
      - Attack via case sensitivity, spaces
      - $<SCRI<SCRIPT>PT>$
  - Whitelist tags+attributes – 70-80%
    - Dangerous tags like ‘style’
  - Rewrite/transform/canonicalize – 90%
    - Broken if ambiguous parsing
    - UTF 7 bug with how greater than ( or less than? ) is encoded

HTML Filtering
- Samy Worm on MySpace exploited filtering vulnerabilities
- Send javascript to client to force correct parsing of html and rebuild the dom tree, rewrite document from newly formed tree

Static and Dynamic vulnerability detection
- Static Analysis
  - Suggests that you can detect vulnerabilities by running something directly on the code without executing it
  - Very recent field, came into the market within the last 4 years
  - Professor Wagner’s simplest description of static analysis
    - Keep a set of predicates about the program, keep track of them at every point in the program
    - $Foo(char* p){$
      1) \[ if(p==null) \]
      2) \[ return; \]
      3) \[ // infer p!=NULL before dereference, must prove over all executions that this holds true \]
      4) \}
• Evaluate predicates in topologically sorted order of these states
• Example above is very simplistic, leaves out:
  ♦ Loops
  ♦ possibly infinite number of predicates
  ♦ points-to analysis ( lose all information about the heap if a pointer into the heap gets written to )
  ♦ data structures
  ♦ virtual methods
  ♦ functions that return values ( simplistic analysis will not have any information about the value returned in the example above if it returned p instead of nothing )

```c
Foo(char* p){
1) if(p==null)
2)   return p;
   // infer p!=NULL before dereference, must prove over all executions that this holds true
3)   *p = 'x'
4)   return p;
}
```

➢ Even worse if recursive ( loops in the graph )
• All static analysis either misses bugs or gives false positives ( or run forever... )
  ♦ Commercial products try very hard to give a 30% false positive rate
  ♦ Quality -> less false positives, more missed bugs
  ♦ Security -> more false positives, try to hit more bugs
• Need to write this logic in arbitrary first order logic
  ♦ Very tough, so most annotation languages are restricted to a smaller set and focus on a specific vulnerability (e.g. buffer overruns)
  ♦ Also, need to annotate all functions. Very costly.
  ♦ Microsoft SAL

➢ Dynamic Analysis
  ▪ EXE
    • Boolean values per bit, reason symbolically about the state at each point of execution
    • Checks if there is any possible input to trigger an error via SAT solver
    • Example:
      ♦ y = x + z
      ♦ boolean variables $\alpha_{x0}...\alpha_{x31} \alpha_{y0}...\alpha_{y31} \alpha_{z0}...\alpha_{z31}$
• generate a new boolean variable for addition via \( \beta_{yi} = \alpha_{xi} \text{ XOR } \alpha_{zi} \)
• Turns out that the number of constraints per instructions isn’t that big and SAT solvers have gotten so good that it works well even on large programs
  - Taint tracking
  - Constraints on taint propagation hard to do
  - Strings are hard to reason about (e.g. long, regex, other complex processing)
    • If you restrict string lengths, you can reason about it in a bitwise manner and use a SAT solver
  ➢ Hybrid analysis: using both symbolic execution and static analysis to try and find dangerous inputs