Automatic Worm Defense (I)

Dawn Song
dawnsong@cs.berkeley.edu

Primer on Internet Worms (I)

- First Instance:
  - Morris worm (1988)
  - Infected 6000 machines (10% of Internet)
  - $10M for downtime & cleanup
- What's a worm?
  - Self-propagating software
  - In contrast to viruses, etc., which requires human intervention for propagation

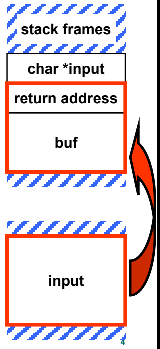
What does it Take to Make a Worm?

- Cause a piece of code to automatically run on a host
  - Exploit a vulnerability (e.g., memory safety)← our focus
  - Can you design worms not exploiting memory safety vulnerabilities?
    - » Morris worm: Rhosts + password guessing
    - » Javascript worms. ← later in class
- Propagate
  - How to find targets to propagate to?
    - » Scan IP addresses
    - » Topological worms

Buffer Overflow

```
int check_http( char *input ) {
  char buf[8];
  if (strncmp(input, "get",3) != 0 &&
      strncmp(input, "put",3) != 0 )
     return -1;
  if (input[3] != '/' ) return -1;
  strncpy( buf, input, 4);
  int i = 4;
  while ( input[i] != '\n')
     { buf[i] = input[i];
        i++; }
  return i;
}
```

Vulnerability condition: i ≥ 8



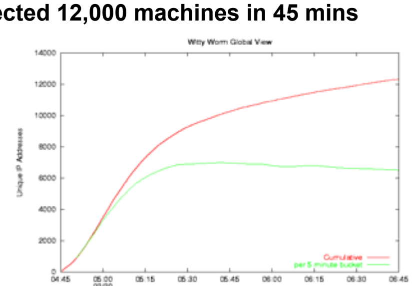
Sample Historical Worms

Worm	Date	Distinction
Morris	11/88	Used multiple vulnerabilities, propagate to "nearby" sys
ADM	5/98	Random scanning of IP address space
Ramen	1/01	Exploited three vulnerabilities
Lion	3/01	Stealthy, rootkit worm
Cheese	6/01	Vigilante worm that secured vulnerable systems
Code Red	7/01	First sig Windows worm; Completely memory resident
Walk	8/01	Recompiled source code locally
Nimda	9/01	Windows worm: client-to-server, c-to-c, s-to-s, …
Scalper	6/02	11 days after announcement of vulnerability; peer-to-peer network of compromised systems
Slammer	1/03	Used a single UDP packet for explosive growth

Kienzle and Elder

Witty Worm (I)

- March 19, 2004, exploiting buffer overflow in firewall (ISS) products
- Infected 12,000 machines in 45 mins



Witty Worm (II)

- First widely propagated worm w. destructive payload
  - Corrupted hard disk
- Seeded with more ground-zero hosts
  - 110 infected machines in first 10 seconds
- Shortest interval btw vulnerability disclosure & worm release
  - 1 day
- Demonstrate worms effective for niche too
- Security devices can open doors to attacks
  - Other examples: Anti-virus software, IDS

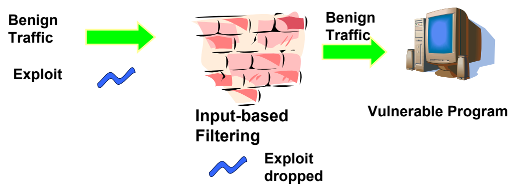
Challenges for Worm Defense

- Short interval btw vulnerability disclosure & worm release
  - Witty worm: 1 day
  - Zero-day exploits
- Fast
  - Slammer: 10 mins infected 90% vulnerable hosts
  - How fast can it be?
    - Flashworm: seconds [Staniford et. al., WORM04]
- Large scale
  - Slammer: 75,000 machines
  - CodeRed: 500,000 machines

Automatic Worm Defense

- Filter/rate-limit based on IP & Port
  - Newly infected IP
  - Huge list
  - IP changes: dynamic IP, etc.
  - NAT
  - Strategy: filter based on *who*
- Filter based on content (a.k.a. input-based filtering)
  - Signatures
  - Can be host-based or network-based
  - Strategy: filter based on *what*
- Why not just patch?
  - Users don't apply patch
  - Patching production systems requires testing
  - Modifying critical systems require re-certification
  - Legacy systems can no longer be patched
  - What to do for zero-day?
  - Dynamic patch ←later in class

Automatic Signature Generation for Input-based Filtering



- **Input-based filtering**
  - **Signature f: given input x, f(x) = exploit or benign**
  - **Effective, widely-deployed defense**
- **Question:**
  How to generate signatures, esp. for new attacks?

Desired Properties for Automatic Signature Generation

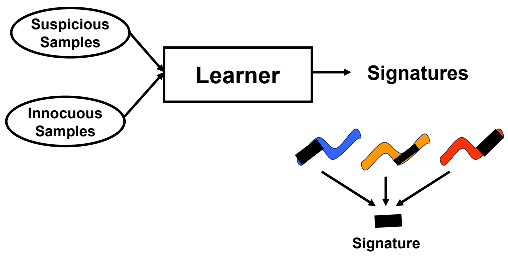
- **Fast generation**
  - **Worm propagates in minutes or seconds**
- **Fast matching**
  - **Low runtime overhead**
- **Accurate**
  - **Low/no false positives**
  - **Low/no false negatives**
  - **Able to measure/guarantee signature quality**
- **Effective against polymorphic worms**

Polymorphic Worms

- **Loose terminology:**
  - **Including polymorphic, metamorphic, etc., techniques**
- **How can you make a worm/exploit polymorphic?**
- **Are there invariants in polymorphic worms?**
- **Key: effective signatures need to identify invariants**

How to Automatically Generate Signatures?

- **Approach I: pattern-extraction based**
  - **Extract common patterns in worm samples, not in benign samples**



13

Pattern-extraction based Signature Generation

- **Honeycomb[Kreibich-Hotnets03]**
  - **Longest common substring**
- **Earlybird[Singh-OSDI03]**
  - **Common substring using Rabin fingerprinting**
- **Autograph[Kim-USENIX05]**
  - **Common substring using content-based payload partitioning**
- **Polygraph[Newsome-IEEE S&P05]**
  - **Combination of common substrings, e.g., conjunctions, subsequences, Bayes,**
  - **Clustering techniques**

14

Disadvantages of Patter-extraction based Signature Generation

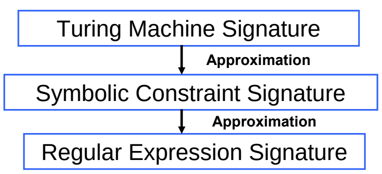
- **Insufficient for polymorphic worms & unseen variants**
- **What kinds of invariants can it discover?**
  - **Depending on the classes of functions learned**
  - **What other functions may be of interest to learn?**
- **No guarantee of signature quality**
  - **How to evaluate signature quality?**
- **Susceptible to adversarial learning [Newsome-RAID06]**
  - **Attackers crafting malicious samples**
  - **How?**
- **Purely bit-pattern syntactic approach, so no semantic understanding of vulnerability**
  - **Only generating exploit-signatures**

15

Approach II: Vulnerability Signature Generation

- **Instead of bit patterns, use root cause**
  - **Generating signatures based on vulnerability**
- **As exploits morph, they need to trigger vulnerability**
- **So, vulnerability puts constraints on exploits**
- **Problem reduction:**
  - **Signature generation = constraints on inputs that trigger vulnerability**
- **Symbolic execution**
- **Soundness guaranteed (no false positives)**

Different Classes of Signatures



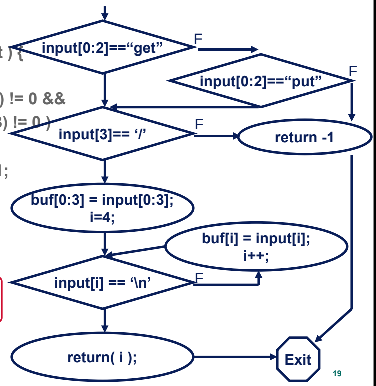
MEP Symbolic Constraint Signatures

- **Monomorphic Execution Path (MEP)**
- **Any input which**
  a) executes same path as exploit &
  b) satisfies vulnerability condition
  is exploit
- **Represent inputs as symbolic variables**
- **Symbolically execute same path as exploit**
  - **Construct symbolic expressions for registers & memory**
- **Signatures = constraint on symbolic input variables**
  - **Conjunctions of branch conditions & vulnerability condition**

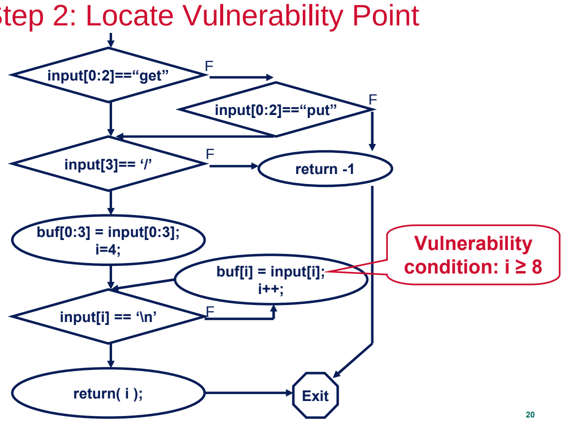
Step 1: Generate Control Flow Graph

```
int check_http( char *input ) {
  char buf[8];
  if (strncmp(input, "get",3) != 0 &&
      strncmp(input, "put",3) != 0 )
    return -1;
  if (input[3] != '/' ) return -1;
  strncpy( buf, input, 4);
  int i = 4;
  while ( input[i] != '\n')
    { buf[i] = input[i];
      i++; }
  return i;
}
```

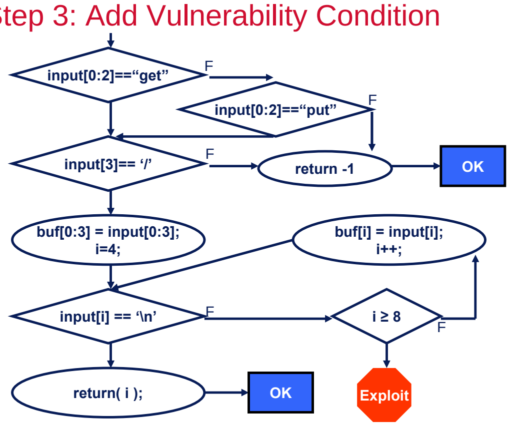
Vulnerability condition: i ≥ 8



Step 2: Locate Vulnerability Point



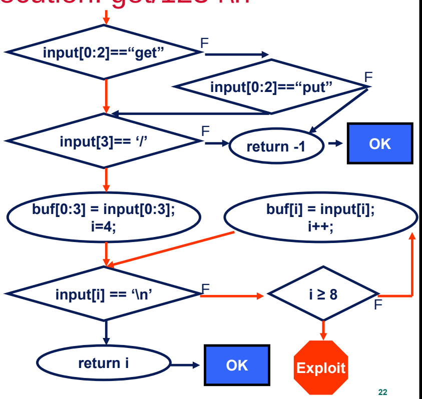
Step 3: Add Vulnerability Condition



Symbolic Execution: get/1234\n

Resulting Constraint:

input[0:2]== ‘get’
& input[3] == ‘/’
& input[4:7] != ‘\n’
& i ≥ 8



MEP Symbolic Constraint Signature

- **Resulting constraint forms**
  MEP Symbolic Constraint Signature

input[0:2]= “get” & input[3] = ‘/’ & input[4:7] != ‘\n’

given x = get/1234\n

- **Signature Accuracy**
  - **Sound:**
    Any input that satisfies the constraint is an exploit
  - **Complete with respect to path:**
    Matches any polymorphic variants along the same path

MEP Regular Expression Signature

- **2nd type of Monomorphic Execution Path Signature**
- **Two subtypes of Regular Expression Signatures:**

1) Under approximation
- **Use a solver (e.g., STP) to solve Boolean formula**
  - » **Automatically generate exploit!**
- **Combine solutions of satisfying assignments by logical OR**
- ➢ **Soundness guaranteed**

2) Over approximation
- **Use a solver to identify range of values of input variables**
- ➢ **Provides a fast first pass:**
  - » **Only check against symbolic constraint signature if matched**

MEP Regular Expression Signature

MEP Symbolic Constraint Signature
input[0:2]= “get” & input[3] = ‘/’ & input[4:7] != ‘\n’

⇓

MEP Regular Expression Signature
get/[^\n][^\n][^\n][^\n]

Limitation for MEP Signatures

- **Only covering a single path**
  - **Different keywords**
  - **Variable length inputs**
  - **Different protocol steps**

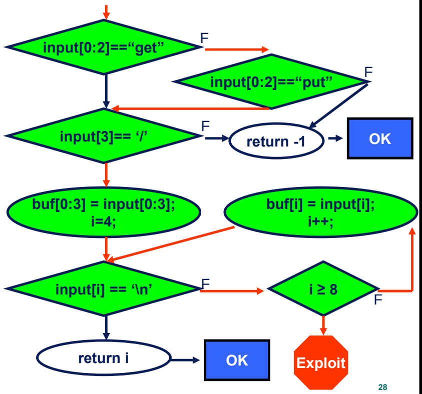
How to Address MEP Limitations?

- **Polymorphic Execution Path (PEP) Symbolic Constraint Signature**
- **Intuition**
  - **Explore different paths to generate additional signatures**
- **Approach I: generating MEP signatures for different paths and combine them**

Different Path

- **Resulting Constraint:**

input[0:2]== 'put'
& input[3] == '/'
& input[4:7] != '\n'
& i >=8



PEP Regular Expression Signature

PEP Symbolic Constraint Signature
input[0:2]= "get" & input[3] = '/' & input[4:7] != '\n'
∨
input[0:2]= "put" & input[3] = '/' & input[4:7] != '\n'

⇓

PEP Regular Expression Signature
[get|put]/[^\n][^\n][^\n][^\n]

Challenges

- **How to pick different paths?**
- **Limitations**
  - **Exponential blow-up in # of paths**
  - **Infinite # of paths due to loops**

What Changes Can You Make to the Program to Demonstrate the Limitations?

```
int check_http( char *input ) {
  char buf[8];
  if (strncmp(input, “get”,3) != 0 &&
      strncmp(input, “put”,3) != 0 )
    return -1;
  if (input[3] != ‘/‘ ) return -1;
  strncpy( buf, input, 4);
  int i = 4;
  while ( input[i] != ‘\n‘)
    { buf[i] = input[i];
      i++; }
  return i;
}
```

Vulnerability condition: $i \geq 8$

Addressing PEP Limitation I

- **Approach II: computing Weakest Precondition [Brumley-CSF07]**
  - Use vulnerability condition as post condition
  - Statically compute weakest precondition over program
    - » With loops unrolled
  - Formula size is polynomial in size of program (unrolled)
  - Challenge: formula size may still be too big
    - » Loops unrolled, functions inlined

Addressing PEP Limitation II

- **Turing Machine signatures**
  - Objective: Generate program to pick path at run time
  - Compute chop between input point and vulnerability point
  - Inline vulnerability condition check at vulnerability point
  - Challenge: difficult to compute precise chop
- **Why Turing Machine (TM) signatures?**
  - Vulnerability language class may require TM signatures for perfect accuracy
  - When may TM signatures be needed in practice?
    - » E.g., need to parse the protocol

Under the Hood

- Implementation works on x86 binary
- Signature generation
  - Convert x86 to Intermediate Language (IL)
  - Symbolic execution + analysis on IL
  - Signature output as C program (or x86 directly)
- Challenges in handling x86 binary
  - Complex instruction set
    - » Implicit arguments (5 operands)
    - » Single instruction jumps
  - Scale
    - » SQL server: more than 3 million LOC in binary; source code orders of magnitude smaller
- Part of BitBlaze project
  - http://bitblaze.cs.berkeley.edu

Impact in Real-world

- Currently applying techniques in Symantec
- Joint venture with Reservoir Labs
- Potential prototype integration with FireEye IPS
- Lots more work to be done

Open Questions

- Can you apply this approach to generate signatures for viruses?
- Are there advantages combining pattern-extraction based/machine learning approaches with PL-based vulnerability singature generation?

Open Mic

- **Questions?**
- **Thoughts you'd like to share**

Summary

- **Automatic signature generation for worm/exploit defense**
  - **Pattern-extraction based techniques**
  - **Vulnerability signature generation**
- **Supplemental reading**
  - **Vigilante**
  - **Shield**
- **Next class:**
  - **How to make vulnerability signature generation practical?**
  - **Other worm/exploit defense mechanisms (if time allows)**
    - **E.g., Dynamic patches**