BitBlaze: Binary Analysis for Computer Security

Dawn Song

Computer Science Dept.
UC Berkeley
Malicious Code---Critical Threat on the Internet

• **Diverse forms**
  – Worms, botnets, spyware, viruses, trojan horses, etc.

• **High prevalence**
  – CodeRed Infected 500,000 servers
  – 61% U.S. computers infected with spyware [National Cyber Security Alliance06]
  – Millions of computers in botnets

• **Fast propagation**
  – Slammer scanned 90% Internet within 10 mins

• **Huge damage**
  – $10billion annual financial loss [ComputerEconomics05]
Defense is Challenging

- Software inevitably has bugs/security vulnerabilities
  - Intrinsic complexity
  - Time-to-market pressure
  - Legacy code
  - Long time to produce/deploy patches
- Attackers have real financial incentives to exploit them
  - Thriving underground market
- Large scale zombie platform for malicious activities
- Attacks increase in sophistication

- We need more effective techniques and tools for defense
  - Previous approaches largely symptom & heuristics based
The BitBlaze Approach

• Semantics based, focus on root cause:

  Automatically extracting security-related properties from binary code (vulnerable programs & malicious code) for effective defense

• Automatically create high-quality detection & defense mechanisms
  – Automatic generation of vulnerability signatures to filter out exploits
  – Automatic detection and classification of malware
    » Spyware, keylogger, rootkit, etc.
    » Automatic detection of botnet traffic

• Able to handle binary-only setting
Binary Analysis: Imperative & Challenging

• Binary analysis is imperative
  – Source code is often unavailable
    » COTS programs
    » Malicious code
  – Binary is truthful

• Binary analysis is challenging
  – Lack higher-level semantics
    » Even disassembling is non-trivial
  – Malicious code may obfuscate
    » Code packing
    » Code encryption
    » Code obfuscation & dynamically generated code

• Need techniques & tools to address these issues
The BitBlaze Vision & Research Foci

1. Design and develop a unified binary analysis platform for security applications
   – Identify & cater common needs of different security applications
   – Leverage recent advances in program analysis, formal methods, binary instrumentation/analysis techniques to enable new capabilities

2. Introduce binary-centric approach as a powerful arsenal to solve real-world security problems
   • COTS vulnerability analysis & defense
   • Malicious code analysis & defense
   • Other security applications
The BitBlaze Binary Analysis Platform

• A unique infrastructure:
  – Novel fusion of static, dynamic analysis techniques, and formal analysis techniques such as symbolic execution
  – Vine: accurate static analysis using VineIL (Intermediate Language)
  – TEMU: whole-system, fine-grained, symbolic emulation system
  – Rudder: automatic exploration of program execution space
BitBlaze in Action: Addressing Security Problems

- Effective new approaches for diverse security problems
  - Over dozen projects
  - Over 12 publications in security conferences
- Exploit detection, diagnosis, defense
- In-depth malware analysis
- Others:
  - Reverse engineering
  - Deviation detection [Best Paper Award]
  - Semantic binary diff
Talk Outline

• Motivating security applications
  – Automatic patch-based exploit generation

• Components
  – Vine: VineIR, static analysis on VineIR
  – TEMU: whole-system, fine-grained, symbolic emulation system
  – Rudder: automatic execution space exploration

• Future directions and conclusion
Automatic Patch-based Exploit Generation

• Given vulnerable program P, patched program P’, automatically generate exploits for P

• Why care?
  – Exploits worth money
    » Typically $10,000 - $100,000
  – Know thy enemy
    » Security of patch distribution schemes?
  – Patch testing
Running Example

- All integers unsigned 32-bits
- All arithmetic mod $2^{32}$
- Motivated by real-world vulnerability

```
read input

if input % 2 == 0

F
s := input + 3
T
s := input + 2

ptr := realloc(ptr, s)
```
Running Example

input = $2^{32}-2$

$s := 0 \cdot (2^{32}-2 + 2 \cdot 2^{32})$

ptr := realloc(ptr, 0)

Using ptr is a problem
Running Example

read input

\[ \text{if } \text{input} \mod 2 == 0 \]

1. \[ \text{s := input + 3} \]
2. \[ \text{s := input + 2} \]
3. \[ \text{ptr := realloc(ptr, s)} \]

Integer Overflow when: \[ s < \text{input} \]
Running Example

I didn’t think about overflow!

Exploits:
\[ 2^{32}-3, \quad 2^{32}-2, \quad 2^{32}-1 \]

All 32-bit integers

Safe inputs
• Programmer fails to sanitize inputs
• Large class of security-critical vulnerabilities
  – “Buffer overflow”, “integer overflow”, “format string vulns”, etc.
• Responsible for many, many compromised computers
if input % 2 == 0

F
s := input + 3

T
s := input + 2

ptr := realloc(ptr, s)

Overflow when s < input

Patch leaks

1. Vulnerability point (where in code)
2. Vulnerability condition (under what conditions)
Exploits for P are inputs that fail vulnerability condition at vulnerability point
(s > input) = false
Our Approach for Patch-based Exploit Generation (I)

**Exploit Generation**

1. **Diff P and P’ to identify candidate vuln point and condition**
2. **Create input that satisfy candidate vuln condition in P’**
   - i.e., candidate exploits
3. **Check candidate exploits on P**

- **Patch**
  - `read input`
  - `if input % 2 == 0`
    - `F`
    - `T`
  - `s := input + 3`
  - `s := input + 2`
    - `if s > input`
      - `F`
      - `T`
        - `Error`
        - `ptr := realloc(ptr, s)`
Our Approach for Patch-based Exploit Generation (II)

- **Diff P and P’ to identify candidate vuln point and condition**
  - Currently only consider inserted sanity checks
  - Use binary diffing tools to identify inserted checks
    » Existing off-the-shelf syntactic diffing tools
    » BinHunt: our semantic diffing tool

- **Create candidate exploits**
  - i.e., input that satisfy candidate vuln condition in P’

- **Validate candidate exploits on P**
  - E.g., dynamic taint analysis (TaintCheck)
Create Candidate Exploits

- Given candidate vulnerability point & condition
- Compute Weakest Precondition over program paths
  - Using vulnerability condition as post condition
  - Construct formulas representing conditions on input
    » Whose execution path included
    » Satisfying the vulnerability condition at vulnerability point
- Solve formula using solvers
  - E.g., decision procedures
  - Satisfying answers are candidate exploits
Different Approaches for Creating Formulas

• **Statically computing formula**
  – Covering many paths (without explicitly enumerating them)
  – Sometimes hard to solve formula

• **Dynamically computing formula**
  – Formula easier to solve
  – Covering only one path

• **Combined dynamic and static approach**
  – Covering multiple paths
  – Tune for formula complexity

• **Experimental results**
  – Different approach effective for different scenarios

• **Other techniques to make formulas smaller and easier to solve**
Experimental Results

- 5 Microsoft patches
  - Mostly 2007
  - Integer overflow, buffer overflow, information disclosure, DoS
- Automatically generated exploits for all 5 patches
  - In seconds to minutes
  - 3 out of 5 have no publicly available exploits
  - Automatically generated exploit variants for the other 2
- Diffing time
  - A few minutes
## Exploit Generation Results

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>DSA_SetItem</th>
<th>ASPNet_Filter</th>
<th>GDI</th>
<th>IGMP</th>
<th>PNG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dynamic Total</strong></td>
<td>5.68</td>
<td>11.57</td>
<td>10.34</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>5.51</td>
<td>4.64</td>
<td>10.33</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Solver</strong></td>
<td>0.17</td>
<td>6.93</td>
<td>0.01</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Static Total</strong></td>
<td>83.47</td>
<td>N/A</td>
<td>26.41</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>2.32</td>
<td>N/A</td>
<td>4.99</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Solver</strong></td>
<td>81.15</td>
<td>N/A</td>
<td>21.42</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Combined</strong></td>
<td>11.51</td>
<td>N/A</td>
<td>29.07</td>
<td>13.57</td>
<td>104.28</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>6.72</td>
<td>N/A</td>
<td>25.29</td>
<td>13.31</td>
<td>104.14</td>
</tr>
<tr>
<td><strong>Solver</strong></td>
<td>4.79</td>
<td>N/A</td>
<td>3.78</td>
<td>0.26</td>
<td>0.14</td>
</tr>
</tbody>
</table>
Talk Outline

• Motivating security applications
  – Automatic patch-based exploit generation

• Components
  – Vine: VineIR, static analysis on VineIR
  – TEMU: whole-system, fine-grained, symbolic emulation system
  – Rudder: automatic execution space exploration

• Future directions and conclusion
Vine

- Static analysis component
Vine IR

• Simple RISC-like language, well-typed

```
lval := exp
| goto exp
| if exp then goto exp₁ else exp₂
| return exp
| call exp
| assert exp
| special exp
| unknown (effects)
```

• Handle x86, and ARM in progress
TEMU

- Work for both Windows & Linux, applications & kernel
- Build on QEMU
Rudder

- Compute path predicate
- Obtain new path predicate by reverting branches
- Solve path predicate to obtain new input to go down a different path
BitScope

- Built on top of TEMU & Rudder
- Work for packed code, self-encrypted code
BitScope: THE In-depth Malware Analysis infrastructure

• Identify/analyze malicious behavior based on root cause
  – Privacy-breaching malware: spyware, keylogger, backdoor, etc.
  – Malware perturbing system by hooking: rootkit, etc.
• Understand how malware get into the system
  – What mechanisms/vulnerabilities does it exploit
• Explore hidden behavior, detect trigger-based behavior
  – Automatically identifying botnet program commands, time bombs, etc.
• Semantic & correlation analysis of malware input/output behavior
  – Understanding the semantics of botnet program commands, etc.
Challenges

• Performance & scalability for large programs

• Sample components we can take advantage of
  – Better identification of functions & resolution of indirect jumps
    » Some of our VSA techniques may help
  – Better stack-walker
  – Binary aliasing analysis
  – More efficient binary instrumentation
Conclusion

• BitBlaze binary analysis platform
  – A unique fusion of dynamic, static analysis & formal analysis (symbolic execution, WP, etc.)

• Security Applications
  – Vulnerability discovery, diagnosis, defense
  – In-depth malware analysis
  – Reverse engineering
  – Binary diffs

• Components may support other applications
Contact

- [http://bitblaze.cs.berkeley.edu](http://bitblaze.cs.berkeley.edu)

- [dawnsong@cs.berkeley.edu](mailto:dawnsong@cs.berkeley.edu)

- **BitBlaze team:**
  David Brumley, Juan Caballero, Ivan Jager, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Prateek Saxena, Heng Yin